Solving dense linear systems on accelerated multicore architectures

Adrien Remy

Université Paris-Sud

Thèse

pour obtenir le grade de

Docteur en Informatique de l'Université Paris-Sud

préparée au Laboratoire de Recherche en Informatique
dans le cadre de *l'École Doctorale 427 : Informatique Paris-Sud*

présentée et soutenue publiquement

par

Adrien Rémy

8 Juillet 2015

Solving dense linear systems On accelerated multicore architectures

---

Directeur de thèse :
Mr. Marc Baboulin

Jury

M. Stef Graillat	Rapporteur	Professeur Université Paris 6
M. Paulo Beleza Vasconcelos	Rapporteur	Professeur Universidade do Porto
M. Philippe Langlois	Examinateur	Professeur Unviversité de Perpignan
M. Marc Baboulin	Directeur de thèse	Professeur Université Paris Sud
M. Nicolas M.Thiéry	Examinateur	Professeur Université Paris Sud

Solving dense linear systems on accelerated multicore architectures

Abstract:

In this PhD thesis, we study algorithms and implementations to accelerate the solution of dense linear systems by using hybrid architectures with multicore processors and accelerators. We focus on methods based on the LU factorization and our code development takes place in the context of the MAGMA library.

We study different hybrid CPU/GPU solvers based on the LU factorization which aim at reducing the communication overhead due to pivoting. The first one is based on a communication avoiding strategy of pivoting (CALU) while the second uses a random preconditioning of the original system to avoid pivoting (RBT). We show that both of these methods outperform the solver using LU factorization with partial pivoting when implemented on hybrid multicore/GPUs architectures.

We also present new solvers based on randomization for hybrid architectures for Nvidia GPU or Intel Xeon Phi coprocessor. With this method, we can avoid the high cost of pivoting while remaining numerically stable in most cases. The highly parallel architecture of these accelerators allow us to perform the randomization of our linear system at a very low computational cost compared to the time of the factorization.

Finally we investigate the impact of non-uniform memory accesses (NUMA) on the solution of dense general linear systems using an LU factorization algorithm. In particular we illustrate how an appropriate placement of the threads and data on a NUMA architecture can improve the performance of the panel factorization and consequently accelerate the global LU factorization. We show how these placements can improve the performance when applied to hybrid multicore/GPU solvers.

Keywords: Dense linear systems, LU factorization, dense linear algebra libraries, MAGMA library, hybrid multicore/GPU computing, graphics process units, Intel Xeon Phi, ccNUMA, communication-avoiding algorithms, randomization, thread placement.

Résoudre des systèmes linéaires denses sur des architectures composées de processeurs multicœurs et d'accélerateurs.

Résumé :

Dans cette thèse de doctorat, nous étudions des algorithmes et des implémentations pour accélérer la résolution de systèmes linéaires denses en utilisant des architectures composées de processeurs multicœurs et d'accélérateurs. Nous nous concentrons sur des méthodes basées sur la factorisation LU. Le développement de notre code s'est fait dans le contexte de la bibliothèque MAGMA.

Tout d'abord nous étudions différents solveurs CPU/GPU hybrides basés sur la factorisation LU. Ceux-ci visent à réduire le surcoût de communication dû au pivotage. Le premier est basé sur une stratégie de pivotage dite "communication avoiding" (CALU) alors que le deuxième utilise un préconditionnement aléatoire du système original pour éviter de pivoter (RBT). Nous montrons que ces deux méthodes surpassent le solveur utilisant la factorisation LU avec pivotage partiel quand elles sont utilisées sur des architectures hybrides multicœurs/GPUs.

Ensuite nous développons des solveurs utilisant des techniques de randomisation appliquées sur des architectures hybrides utilisant des GPU Nvidia ou des coprocesseurs Intel Xeon Phi. Avec cette méthode, nous pouvons éviter l'important surcoût dû pivotage tout en restant stable numériquement dans la plupart des cas. L'architecture hautement parallèle de ces accélérateurs nous permet d'effectuer la randomisation de notre système linéaire à un coût de calcul très faible par rapport à la durée de la factorisation.

Finalement, nous étudions l'impact d'accès mémoire non uniformes (NUMA) sur la résolution de systèmes linéaires denses en utilisant un algorithme de factorisation LU. En particulier, nous illustrons comment un placement approprié des processus légers et des données sur une architecture NUMA peut améliorer les performances pour la factorisation du panel et accélérer de manière conséquente la factorisation LU globale. Nous montrons comment ces placements peuvent améliorer les performances quand ils sont appliqués à des solveurs hybrides multicœurs/GPU.

Mots clés : Systèmes linéaires denses, factorisation LU, bibliothèques logicielles pour l'algèbre linéaire dense, bibliothèque MAGMA, calcul hybride multicœur/GPU, processeurs graphiques, Intel Xeon Phi, ccNUMA, communication-avoiding, randomisation, placement des processus légers.

Contents

List of Figures

Introduction

In many computational applications, the most time and resource consuming task consists in solving a linear system of equations of the form $Ax = b$. Then the major challenge is to compute a solution x as fast as possible while maintaining a satisfactory accuracy. The main purpose of this PhD thesis is to study solutions to accelerate dense linear solvers using state-of-the-art parallel architectures, which often include accelerators.

In this work we focus on the LU decomposition of dense matrices. We propose different algorithms and implementations to accelerate this factorization.These dense solvers can be used to directly solve systems of equations or as kernels in sparse direct or iterative solvers.

To solve these problems as fast as possible, the algorithms should be adapted to be efficient and scalable on current parallel machines. Moreover the solver implementations should be adapted to the architectural features of these parallel systems. In our work we take into account several characteristics of parallel computers: the use of accelerators such as GPGPUs and Intel Xeon Phi coprocessors, the SIMD parallelism required to program efficiently these accelerators, and Non Uniform Memory Access (NUMA) architectures used in multi-socket shared memory computers.

These parallel architectures provide an increasing computational power and need some special requirements to be exploited efficiently. For example to take advantage of accelerators, we need to consider relatively large problems with a high arithmetic intensity. Also we need to take into account the cost of data transfers between the host and the accelerator through the PCI Express bus which has a limited bandwidth capacity. The use of the SIMD programming requires memory alignment and a very low level programming paradigm. The NUMA architectures require a good management of data locality and thread placement to avoid congestion on the memory links.

Our code developments are made in the context of the MAGMA library which is a dense numerical linear algebra library, designed for hybrid architectures with accelerator (including GPU and Intel Xeon Phi). MAGMA implements the algorithms of the widely used LAPACK library.

We propose different algorithms and implementations to solve large dense linear systems of equations via the LU factorization and some of the resulting code have been integrated into the MAGMA library. We compare the performance of these new optimized implementations to the state-of-the-art methods and discuss their numerical stability.

In Chapter 1, we present an overview of the scientific background of this thesis. We first discuss requirements for solving linear systems of equations and the historical connection with the evolution of computers. We mention the numerical stability issues and explain what are the state-of-the-art solutions used for maintaining sta-

bility thanks to pivoting. We present details on the methods used to solve dense linear systems and emphasize on the LU decomposition, describing the methods of pivoting and parallel algorithms. We then describe the parallel architectures used for High Performance Computing (HPC), discussing their pros and cons and how they can be used in this work. This thesis being closely related with the development of dense linear algebra libraries, we describe the evolution of these libraries and their applications.

In Chapter 2, we discuss different methods of pivoting in the LU factorization algorithm, when performed on hybrid architecture combining multicore processors and GPUs. We describe how the LU factorization is implemented in the MAGMA library and we explain how it can be adapted for different pivoting strategies. We present our implementation of the Communication-Avoiding LU (CALU) factorization for hybrid architecture with GPU and we give details on factorization using multiple GPUs. We then compare the numerical stability and provide performance results.

In Chapter 3, we focus on the use of Random Butterfly Transformation (RBT), in linear system solvers using accelerators. We describe our implementations of the RBT solver for the MAGMA library. We describe the methods we used and how in particular we take advantage of the GPU and Intel Xeon Phi accelerators. We also give some details on the randomization cost and provide performance results.

In Chapter 4, we study and compare different methods to use efficiently Non Uniform Memory Access (NUMA) platforms in the context of dense linear algebra libraries. We propose different methods of thread and data placement to ensure data locality. We apply these methods to the LU factorization comparing their impact on the performance for the panel factorization of the solver and on a global hybrid solver using a GPU as an accelerator.

We finally give some concluding remarks and discuss some ongoing or possible research tracks.

Chapter 1

Algorithms, architectures and libraries

Contents

1.1 Introduction

Solving linear systems of equations has always been a valued approach to solve real life problems in numerous domains such as physics, biology, geometry... Four

thousand years ago, Babylonians had already found out how to solve a 2×2 linear system. Around 200 BC, a Chinese book called the "Nine chapters of mathematical art" explained how to solve a 3×3 linear system using a method similar to the Gaussian elimination [2].

However the study of linear algebra began in the late 17^{th} century with the study of determinants by Gottfried Leibniz. In the early 19^{th} century, Carl Gauss developed a method called "Gaussian elimination" in order to solve linear systems of equations. Then in 1848, James Joseph Sylvester introduced the word "matrix" and in 1855 Arthur Cayley defined the matrix multiplication. Matrix computations took a turning point around World War II with the emergence of the first computers. This allowed linear algebra methods to solve faster and more accurately large systems of equations. Note that Gaussian elimination is still the best known method to solve a linear system of equations [3].

Then it was possible to solve bigger systems thanks to the development of computers. In 1941, Konrad Zuse designed the Z3 machine, an electromechanic computer, the first electrically driven to use the binary system. Two years later in 1943 was built Colossus which was used during World War II to decipher communications between German officers using the Lorenz cipher. During the same period IBM developed the Harvard Mark I which was the first to be fully automatic. It was used in the Manhattan project to run simulations in the development of the first atomic bomb [4]. In 1948, ENIAC was the first computer designed to be Turing complete, it used vacuum tubes. The same year, the Small-Scale Experimental Machine (SSEM) was the first to be based on the von Neumann architecture which uses a single memory to store the instructions and the data [5].

During the 1950s, transistors, which are much more smaller and more reliable, replaced vacuum tubes in the architectures. During this period, technological breakthrough such as the creation of the microcode and the implementation of the first high level language: Fortran, helped to spread the use of computers in scientific and commercial applications.

The integrated circuit, first produced by Texas Instruments in 1958, would soon be used in computers, for example in the Apollo guidance computer in 1963. One year later IBM announced the IBM 360, that was the first system to be based on integrated circuits. The smaller size and cheaper cost of the integrated circuits-based computer placed them as the new standard of computers.

In 1971, Intel released the 4004, the first commercial microprocessor, uniting all the elements of a processor into a single chip. This processor was slow and contained a relatively small number of transistors, but the evolution of the microprocessors would then follow the prediction of Gordon Moore: their complexity will double every year [6].

In 1976 was introduced the Cray 1, one of the first supercomputers to use vector processors in order to accelerate computations.

Since then, architectural evolution helped to build more efficient processors: Instruction pipelines allow instructions to be streamed and reordered, cache memories speed up memory access, branch predictors improve the pipeline, processor with

multiple cores allow local parallelism.

In supercomputers, the number of processors used in parallel increased to thousands and new solutions like accelerators (GPGPU or Intel Xeon Phi) are developed to enhance performance.

To take advantage of these architectural developments, software libraries were released to give the user the possibility to perform efficient linear algebra computations on these architectures. Already with the IBM 360, IBM proposed in 1968 the Scientific Subroutine Package. In 1979, the Basic Linear Algebra Subprograms (BLAS) set of subroutines allowed to perform common linear algebra operations. The same year, LINPACK used BLAS to provide a software library able to perform numerical linear algebra operations on vector-computers. LAPACK appeared as an alternative in 1992, providing optimized routines for cache based architectures. LAPACK had different evolutions during the years, among them ScaLAPACK for distributed architectures, PLASMA for multicores, and MAGMA for hybrid architectures using accelerators like GPGPUs or Intel Xeon Phi.

In this thesis we propose solutions to use or improve some of the current public domain linear algebra libraries so that they exploit the possibilities of modern parallel architectures at their best. We implement different algorithms that are more adapted to some of these architectures to achieve better performance. We also propose methods to optimize memory access in the case of NUMA architectures.

This chapter presents the background of our work. We first review the main issues in solving dense linear systems and we introduce the main methods for solving these systems. We then present the LU factorization and its characteristics, showing the importance of numerical stability and presenting different methods of pivoting. We also present different existing block algorithms and parallel strategies for pivoting in the context of the LU decomposition, focusing on the Communication Avoiding LU factorization (CALU). Next we present the evolutions in parallel architectures and the recent trends in this domain.

In the last part of this chapter we present the development progress in dense numerical algebra libraries.

1.2 Solving dense linear systems

Large dense linear systems are encountered in various scientific fields such as:

Electromagnetics: when problems are solved using Boundary Integral Equations using the "Moment Methods" in the context of the Helmholtz equation. This arises for example in the military domain for stealth airplane technology.

Fluid mechanics: also to solve Boundary Integral Equations but using the "Panel Methods" in the context of the Laplace equation. This is used to understand and model the flow of a fluid passing an object, with applications in aeronautics, construction, etc.

Quantum mechanics: As expressed in [7]: "In quantum mechanical scattering, the goal is to predict the probability that a projectile will scatter off a target with some specific momentum and energy." In [8] they had to solve dense linear systems with 6500 unknowns and they foresee the need to solve systems with 100000 unknowns.

Dense linear systems are also encountered for example in tomography, for noise reduction or for supercomputer benchmarking [9]. Moreover, routines for dense linear systems are commonly used as kernels in more general methods for solving sparse linear systems using direct or iterative methods [10, 11, 12].

Solving these problems generally consists of solving a linear system of equations: $Ax = b$. Thus our goal is to solve such systems as efficiently and accurately as possible.

To solve such systems, there are two classes of methods: direct and iterative methods. The direct methods use a finite sequence of operations to solve the problem while the iterative methods use an initial guess of the result, and generate approximate solutions and tries to make the iterations converge. Iterative methods can be interesting if the system is large and sparse. In our work we are concerned with dense matrices and we focus on direct methods.

Direct methods generally involve the decomposition of matrices followed by the successive resolution of triangular systems. Different methods of decomposition exist such as QR factorization, Cholesky, LDL$^{\mathrm{T}}$ or LU [13].

LU: is used to solve general systems and decompose a matrix A in a product of a unit lower triangular matrix, L and an upper triangular matrix, U. It requires about $2/3 \times n^3$ floating point operations (flops).

LDL$^{\mathbf{T}}$: is used for symmetric matrices. A symmetric matrix A is decomposed as follows, $A = LDL^T$ where L is a unit lower triangular matrix and D a diagonal (or block-diagonal) matrix. It requires about $n^3/3$ flops.

Cholesky: for symmetric positive definite matrices (i.e. all the eigenvalues of the matrix are positive). A is factored as $A = GG^T$ with G a lower triangular matrix with positive diagonal entries. It requires about $n^3/3$ flops.

QR: to solve full rank least square problems in the case of overdetermined systems (the system has more equations than unknowns). A m by n matrix A is factored as $A = QR$ with Q being an m by m orthogonal matrix and R an m by n upper triangular matrix. It requires about $2n^2(m - n/3)$ flops.

In the remainder, we focus on the LU decomposition.

1.3 LU factorization

1.3.1 Gaussian elimination

If A is square, dense and unstructured, the method usually chosen to solve $Ax = b$ is Gaussian elimination. Gaussian elimination consists of a sequence of basic operations on the rows of the matrix to fill the coefficients under the diagonal with zeros making the matrix upper triangular, and thus allowing the system to be solved easily. The LU factorization is a modified form of Gaussian elimination where the matrix A is expressed as a product LU, with L a unit lower triangular matrix and U an upper triangular matrix.

Then the system is easy to solve by forward substitution for L and backward substitution for U:

$Ly = b, Ux = y \Rightarrow Ax = LUx = Ly = b$. This requires $\mathcal{O}(n^2)$ flops.

We compute L and U such as $A = L \times U$ as in the following example:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \text{ and } U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

Algorithm 1 shows how the LU factorization can be performed in place which means that the input matrix A is overwritten by the output factors L and U.

Algorithm 1 In place LU factorization without pivoting

Input: A is a $n \times n$ matrix

```
1: for k ← 1 to n − 1 do
2:   for i ← k + 1 to n do
3:     A(i, k) ← A(i, k)/A(k, k)
4:   end for
5:   for i ← k + 1 to n do
6:     for j ← k + 1 to n do
7:       A(i, j) ← A(i, j) − A(i, k) ∗ A(k, j)
8:     end for
9:   end for
10: end for
```

1.3.2 The issue of pivoting

With the method described previously, if a 0 is found on the diagonal of the matrix, a division by zero will occur and the factorization will fail. Also if elements of small magnitude are on the diagonal, entries on the triangular factors will grow significantly. Moreover the numerical precision is limited on a computer, when finite precision arithmetic is used. Consequently rounding errors are unavoidable. These errors due to limited precision will be propagated and amplified by the division by very small values. The larger systems are more prone to rounding errors.

The stability of the Gaussian elimination can be measured by the growth factor which measures how large the entries of the matrix become during the elimination steps comparing to the largest entries of the input matrix. The growth factor of a $n \times n$ matrix A under Gaussian elimination is defined as:

$$g_n(A) = \frac{\max_{i,j,k} |a_{ij}^{(k)}|}{\max_{i,j} |a_{i,j}|},$$

where $a_{ij}^{(k)}$ is the element of index (i, j) after the step number k of the elimination [14].

For this reason we move the largest element of the column on the diagonal by swapping rows. This method is called partial pivoting. It is also possible to swap rows and columns, using the largest element of the matrix (complete pivoting), or of the current line and column (rook pivoting), as the pivot. Other parallel pivoting strategies will be addressed later.

An alternative method is the threshold pivoting which consists in choosing any pivot among the column if this pivot's absolute value is larger than a predetermined threshold value chosen in $]0, 1]$. First introduced in the context of sparse matrix computations [15, Chapter 5.4], it can be used for dense matrices as shown in [16].

Even though pivoting increases the stability and requires no additional floating point operations, it involves irregular data movements due to the comparisons performed in the process of finding the pivot. If n is the size of the matrix, complete pivoting requires $\mathcal{O}(n^3)$ comparisons, partial pivoting $\mathcal{O}(n^2)$ comparisons and rook pivoting between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$. Therefore, even if complete pivoting has the best stability with a growth factor bound of $cn^{1/2}n^{1/4\log n}$ [17] comparing to rook pivoting ($1.5n^{3/4\log n}$) [18] and partial pivoting (2^{n-1}) [17], it will be time-consuming, due to the comparisons and data movements. The choice of the pivoting strategy is then the result of a compromise between the numerical stability and the performance.

1.3.3 Partial pivoting

In the following, we consider partial pivoting, described in algorithm 2.

The growth factor upper bound is 2^{n-1} can be reached for certain problems [19]. However, partial pivoting is stable in practice.

Algorithm 2 In place LU factorization with partial pivoting

Input: A is a $n \times n$ matrix

```
1: for k ← 1 to n do
2:    index ← k
3:    for i ← k + 1 to n do
4:       if |A(i, k)| > |A(index, k)| then
5:          index ← i
6:       end if
7:    end for
8:    swap rows k and index
9:    for i ← k + 1 to n do
10:      A(i, k) ← A(i, k)/A(k, k)
11:   end for
12:   for i ← k + 1 to n do
13:      for j ← k + 1 to n do
14:         A(i, j) ← A(i, j) − A(i, k) ∗ A(k, j)
15:      end for
16:   end for
17: end for
```

The decomposition is of the form $PA = LU$, where P is the permutation matrix corresponding to the row swaps performed during the factorization.

1.3.4 Block LU factorization

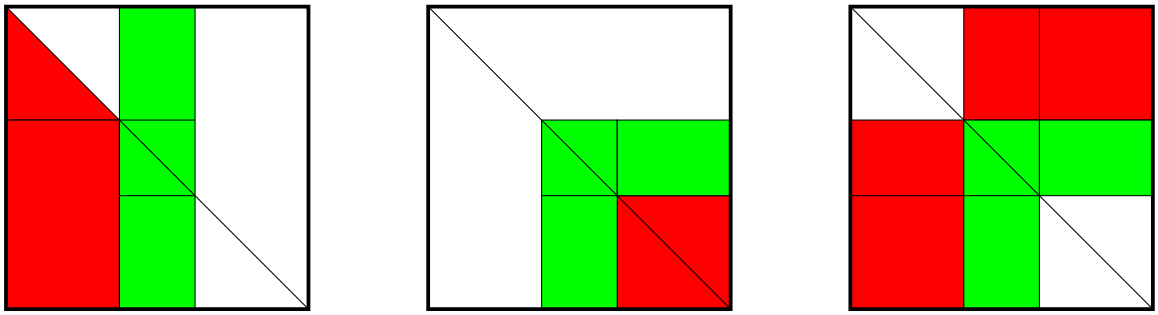
To allow parallelism and a more optimal use of hierarchical memory, we can organize the LU factorization so that matrix multiplications become the dominant operations. For that we perform the factorization by block.

The three common algorithms for the block LU factorization are left-looking LU, right-looking LU and Crout LU.

- The left-looking variant in Figure (1.1a) consists, for each step, in computing a block column using the previously computed ones.
- The right-looking variant in Figure (1.1b) computes for each step a block of rows and columns and then updates the trailing submatrix.
- Crout in Figure (1.1c) is a hybrid version in between left and right looking, a block row and a block column are computed on each step using the previously computed rows and columns.

These variants, due to the arrangements of the loops are also called i, j, k variants. More details can be found in [20].

Legend in the figure 1.1: previously computed parts, on going parts, left to compute.



(a) Left-looking LU (b) Right-looking LU (c) Crout LU

Figure 1.1: Memory access patterns for variants of LU decomposition from [1]

1.3.5 Right-looking block LU

We describe here the right-looking block LU factorization.

We compute the factorization of a matrix A of size $m \times n$. The matrix A is partitioned as follows ,

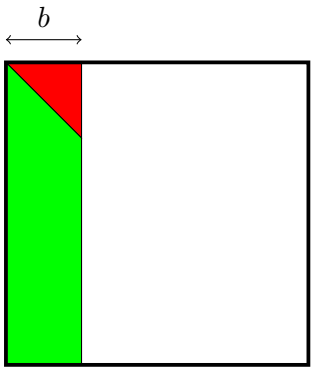
$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where A_{11} is of size $b \times b$, A_{21} is of size $(m-b) \times b$, A_{12} is of size $b \times (n-b)$ and A_{22} is of size $(m-b) \times (n-b)$.

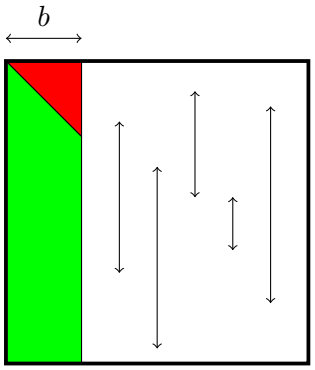
The right looking LU factorization involves 4 steps:

Legend: original matrix, L matrix, U matrix, updated matrix,

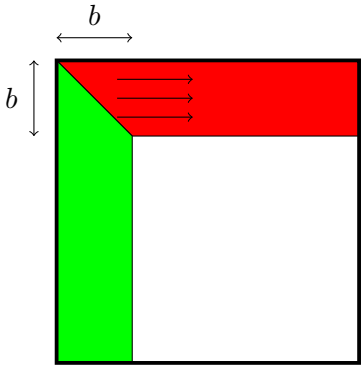
1. The LU factorization of the panel $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ is computed by applying a Gaussian elimination with partial pivoting (GEPP) to $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ (the pivot selection is done on the whole panel)



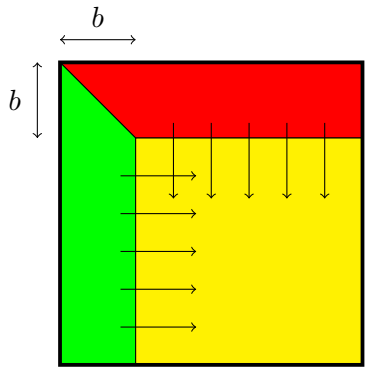
2. We apply the permutation to the rest of the matrix: $\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$



3. We compute U_{12} by solving the triangular linear system $U_{12} = L_{11}^{-1} A_{12}$



4. We update A_{22}: $A_{22} = A_{22} - L_{21}U_{12}$



5. We apply the same method to A_{22} (yellow part)

1.3.6 Parallel pivoting strategies

The design of efficient parallel algorithms for the LU factorization requires other strategies of pivoting than the ones described in section 1.3.2.

A pivoting strategy called pairwise pivoting is studied in [21]. This method consists in selecting the largest element in the column pairwise to annihilate the smaller in magnitude values to triangularize the matrix. This method requires $2n-3$ steps, in which a maximum of $n/2$ independent transformations are performed, allowing parallelism. The parallelism pattern can be represented by a reduction tree. The growth factor upper bound of this method is 4^{n-1} but this pivoting strategy is shown to be stable in practice.

Derived from pairwise pivoting, incremental pivoting was introduced in [22]. This method divides the panel into tiles, factors the diagonal tile using partial pivoting and then eliminates the subdiagonal tiles pairwisely. Contrary to partial pivoting it does not factor a complete block column at a time. The matrix is divided into tiles and first the diagonal tile is factored using GEPP. Then this tile is combined with the tile below and factored again. A new factored diagonal tile is obtained and combined with the next tile below and the operation is repeated until the bottom of the matrix is reached. At each step the tiles on the right of the tile being factored with the diagonal tile are updated according to the operations performed on the panel tile. In this way the updates of the submatrix can be performed in parallel and in the same time as the panel factorization.

1.3.6.1 Communication avoiding technique

On parallel architectures, searching the pivot in the block LU decomposition generates a large volume of data movements for which the communication time may be longer than the effective computing time, if the computation does not overlap the communication.

By reducing communication to its minimum, it is possible to achieve better performance despite a larger number of floating point operations.

In [23], the authors proposed the so-called communication avoiding algorithms for some matrix factorizations such as LU and QR. We will focus more specifically on the CALU algorithm (Communication Avoiding LU) as described in [23, 24, 25, 26]. This algorithm proposes a new strategy for selecting the pivot. This algorithm minimizes communication while keeping the numerical stability of GEPP in practice [26].

The particularity of this algorithm is mainly the factorization of the panel, the other steps of the block decomposition are identical to the right looking LU (see 1.3.5). The factorization of the panel is performed using the TSLU (Tall Skinny LU) algorithm also described in [23, 24, 25, 26].

The TSLU algorithm

TSLU is a parallel algorithm that computes the LU factorization of a $m \times b$ matrix with $m \gg b$. The matrix is distributed over P processors following a row-wise block cyclic distribution. The preprocessing step is performed as an all-reduction operation: a tree of GEPP factorizations.

Below are the steps of the algorithm:

1. Each thread performs a local LU factorization of the $m/P \times b$ block-rows (b is the size of the block) that it owns. This is not performed in place so it requires an extra storage of size $m \times b$ for the resulting matrix and a vector of size m for the permutation vector.

2. The threads copy the b pivot rows of their decomposition in the work matrix.

3. Half of the threads perform the same operation on the matrix composed by the two matrices built with the pivots lines found at the previous step, stacked one upon the other.

4. Whenever we reach the root of the reduction tree, we have the b pivot rows.

5. The permutation is applied to the original matrix to have the previously found pivot rows in first positions.

6. The Gaussian elimination without pivoting is performed on the $m \times b$ columns.

The binary tree representing the work done by the threads (here four) can be seen in Figure 1.2 and an example of the execution in Figure 1.3.

In this example with 4 threads, the matrix is distributed over, P_0, P_1, P_2 and P_3. Each of them computes the GEPP on their $m/4 \times b$ working matrix so that they can find the b pivot rows from this part of the matrix. They copy the pivot rows from the original matrix to the working matrix, then P_0 computes GEPP (using LAPACK function) on a matrix composed of the pivot rows from P_0 stacked on the pivot rows from P_1, this is done identically by P_2. At the final step of the preprocessing, P_0 does the same computing on the $2b$ pivot rows resulting from the previous steps.

A more detailed version of the algorithm can be found in [25, 26].

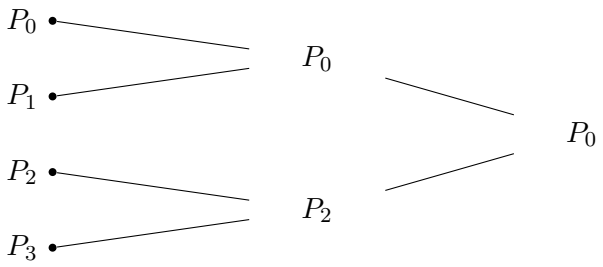


Figure 1.2: Threads work

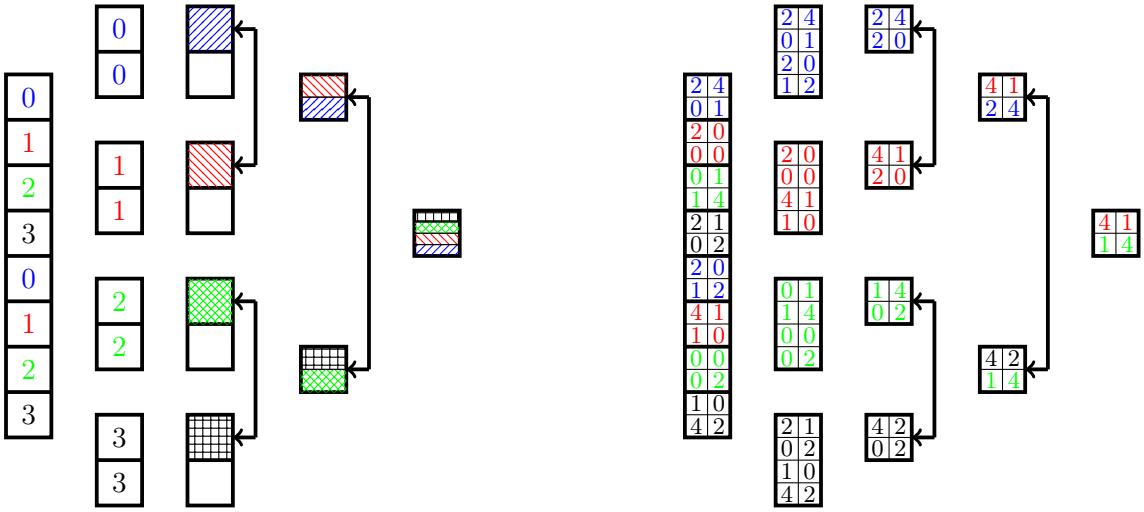


Figure 1.3: Example of execution with 4 threads

1.3.7 Random Butterfly Transformation (RBT)

To avoid pivoting or extra computation that comes along as for example in CALU, it is possible to randomize the matrix and then factorize it using Gaussian Elimination without pivoting. By making the matrix "random enough" the probability that pivoting is not needed will be close to 1. The statistical properties for the stability of Gaussian elimination without pivoting have been studied in [27]. The method for randomizing has been described in [28, 29]. It consists in multiplying a matrix A as $A_r = U^T AV$, where U and V are recursive butterfly matrices. A butterfly matrix is a matrix of the form ,

$$B^{<n>} = \frac{1}{\sqrt{2}} \begin{pmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{pmatrix},$$

where n is the size of the matrix ($n \geq 2$) and R_0 and R_1 are two random diagonal and non singular $n/2 \times n/2$ matrices. A recursive butterfly matrix of size n and depth d is defined recursively as,

$$W^{<n,d>} = \begin{pmatrix} B_1^{<n/2^{d-1}>} & & \\ & \ddots & \\ & & B_{2^{d-1}}^{<n/2^{d-1}>} \end{pmatrix} \times W^{<n,d-1>},$$

where $W^{<n,1>} = B^{<n>}$ and the $B_i^{<n/2^{d-1}>}$ are random butterflies matrices, and $B^{<n>}$ is a butterfly matrix of size n. Then we can perform the LU factorization of A_r using Gaussian Elimination without pivoting. Explanations of how the random butterfly transformations change the growth factor can be found in [28]. Solving the general linear system follows these steps:

1. A randomized matrix A_r is computed: $A_r = U^T AV$ where U and V are recursive random butterfly matrices.
2. A_r is decomposed into LU using Gaussian elimination without pivoting.
3. The system is solved using:$A_r y = U^T b$.
4. The solution is $x = Vy$.

In [28] Parker uses recursive random butterfly of depth $d = \log_2 n$ but in [30] Baboulin et al. showed that in practice a depth of 1 or 2 is enough if iterative refinement is added. Iterative refinement is a method that allows to improve a computed solution of a linear system. If we try to solve a system $Ax = b$, we obtain a computed solution $\hat{x}$. Then the process consists of the 3 following steps (see e.g. [31]):

1. Compute $r = b - A\hat{x}$.
2. Solve $Ad = r$.

3. Update $y = \hat{x} + d$.

The process is repeated with $\hat{x}$ replaced by y until the accuracy of the computed solution is satisfactory.

Due to their particular sparse structure, the butterfly matrices $B^{<n>}$ can be stored in a vector of size $2n$ and the recursive butterfly matrices $W^{<n,d>}$ can be stored in a matrix of size $n \times d$. With this structure the computational cost to apply the multiplicative transformation $(U^T AV)$ is $4dn^2$ flops when U and V are recursive random butterfly matrices of depth d. In the case of PRBT of depth 1 or 2 the computational cost will be respectively $4n^2$ and $8n^2$ [30]. When n is not a multiple of 2^d, we "augment" the matrix A with additional 1s on the diagonal.

Some preliminary results in [30] using CPU to perform the randomization and an hybrid CPU/GPU code to perform the LU factorization without pivoting showed promising result with a 20% gain of performance for matrices of size varying from 4000 to 8000.

1.4 Parallel architectures

In the domain of modern high performance computing (HPC), parallelism is a critical issue. In this section we present an overview of the current architectural solutions developed for HPC.

Since 1945 the Von Neumann architecture is used as a model for building computers. This model subdivides a processing unit into four parts: an arithmetic logic unit, a control unit, a memory which contains data and instructions, and input/output devices [5]. The separation of these elements allows to exploit different types of parallelism. The pipeline was one of the first step toward parallel machines. An early example of pipelined computer is the UNIVAC I (1951) which was able to overlap program execution with some I/O activities. Heavily pipelined processors started with the IBM system/360 Model 91, which was one of the first to use a hierarchy of pipelines [32]. Supercomputers from the 70's to the 90's were mostly designed with vector processors (e.g., the Cray platforms). Different types of parallel architectures exist. The Flynn taxonomy proposes four categories of architectures [33]:

SISD: (Single Instruction, Single Data) where a datum is processed by a single process unit.

MISD: (Multiple Instruction, Single Data) where a single datum is processed by multiple process units at the same time.

SIMD: (Single Instruction, Multiple Data) where multiple data are processed by a single process unit.

MIMD: (Multiple instruction, Multiple Data) where multiple data are processed by multiple process units.

The MIMD model is also completed by Johnson [34] to differentiate the shared memory and distributed memory systems. In the following, we present the architectural components that we used during this PhD thesis.

1.4.1 Distributed memory systems

Many of the current supercomputers are based on distributed memory systems. A distributed memory system consists of multiple independent nodes connected by a given network. Each node has its own private memory and autonomous computational capabilities. The nodes connected together form a cluster. The nodes exchange data by passing messages between processors using the network. Each node can be composed of multiple CPUs and contains accelerators.

1.4.2 SIMD extensions

Single instruction multiple data (SIMD) extensions also called multimedia extensions were introduced in the processors in the late 90's. They provide special registers that can store multiple data. Then instructions can be applied to these registers,

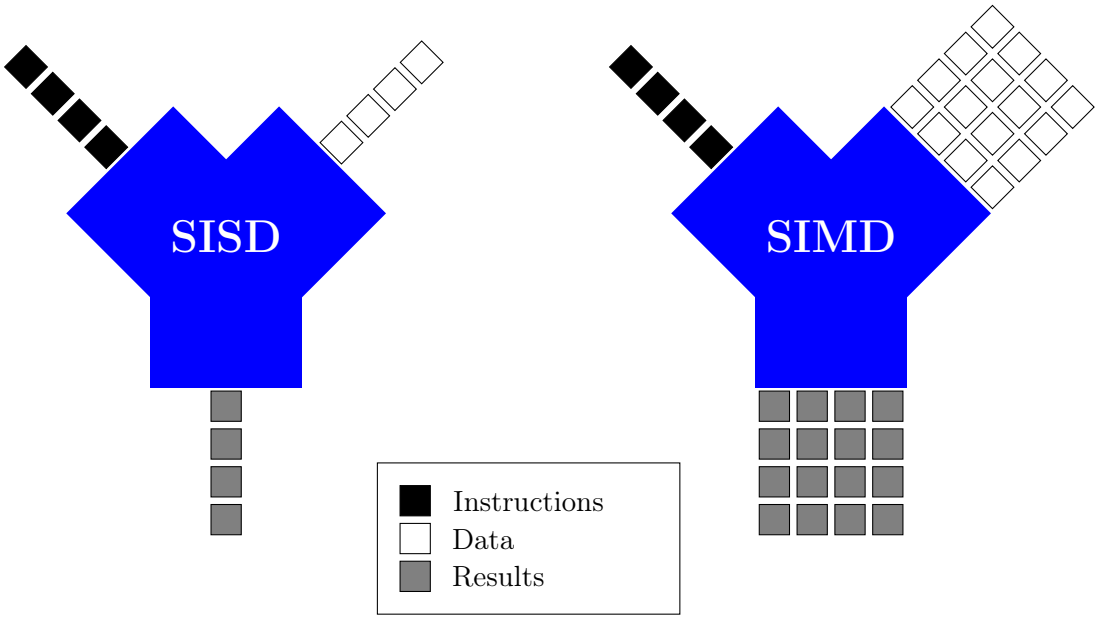


Figure 1.4: Principle of the SIMD extensions.

processing each element of data inside a register simultaneously, which creates parallelism. Figure 1.4 shows the principle of the SIMD extensions, when compared to the SISD model.

SIMD instructions were first used in vector supercomputers, which could apply a single instruction to a vector of data but one word at a time. In 1994, Hewlett-Packard introduced the Multimedia Acceleration eXtensions (MAX) for the PA-RISC instruction set [35] and Sun Microsystems the Visual Instruction Set (VIS) as an extention for the SPARC V9 instruction set [36]. They paved the way for other companies to design their own extensions like Intel with the MMX extension [37] and Motorola with Altivec [38]. MMX can only process integers and uses the float registers to store the values, not allowing to use the SIMD extensions in parallel with scalar computation. Intel released the Streaming SIMD Extensions (SSE) in 1999 for the Pentium III to overcome these limitations. Since then, manufacturers proposed bigger and bigger registers and a larger set of instructions for their SIMD extensions. For example, today's Intel Xeon Phi coprocessor uses AVX-512 that can process 8 double-precision or 16 single-precision floating-point numbers at the same time.

In the domain of dense linear algebra, exploiting the parallelism offered by the SIMD extensions is critical to obtain optimal performance. Most basic operations on matrices or vectors exist in vectorized versions in different implementations of the BLAS libraries such as Intel MKL [39] (more details on these libraries will be given in Section 1.5).

In our work, we used SIMD low level instructions to implement some random-

ization techniques as it will be detailed in Chapter 3.

1.4.3 Multicore processors

On single core processors, manufacturers have developed architectural optimizations such as providing more cache memory, instruction set pipelines and SIMD extensions. The other way to improve the performance of a processor is to increase its frequency. Increasing the frequency is very advantageous for the user because it improves the performance of the programs executed proportionally to the frequency raise. On the other side, for the manufacturer, increasing the frequency rapidly becomes a problem because of the heat generation and the power consumption. The power consumed by a CPU is $P = CV^2f$, with C being the capacitance, V the voltage and f the frequency [40]. The power consumed is proportional to the frequency and the more power is consumed the more thermal power will be produced because of power leakage. The solution favored by manufacturers to continue to improve processor performance without increasing the processor frequency was to introduce multicore processors [41].

IBM developed the POWER4 processor in 2001, the first "on chip" multicore processor. It contains two cores at 1GHz and a shared L2 cache memory [42]. It was followed by SUN and HP releasing respectively the UltraSPARC IV and the PA-8800, both using two cores. AMD and Intel produced the first x86 multiprocessors in 2006 with the AMD Opteron and the Intel Core architectures. Since then, "multicore processor" has become a standard for desktops, servers or mobile platforms. On the desktop market AMD provided 8 core processors working at a frequency up to 4.7 GHz (AMD FX 9590), Intel with the I7-4960X (similar to the architecture depicted in Figure 1.5) provides a 6 core (12 threads with hyper-threading) processor working at up to 4 GHz. On the server side, the AMD Opteron 6386SE provides 16 cores at 2.8 GHz and the Intel Xeon E7-8890v2 15 cores (30 threads) at 2.8 GHz also. Even current smartphones contain processors like the Snapdragon 800 by Qualcomm containing 4 cores based on the ARM V7 architecture up to 2.5 GHz.

Apart from the architecture of the cores used, a multicore processor can be described by [43]:

- The number of processor cores on the chip,
- The number of levels of cache memory,
- The amount of cache memory shared.

1.4.4 Non Uniform Memory Access (NUMA) architecture

Since 1968, computers have been build with multiple processors to perform parallel processing [44]. These machines called symmetric multiprocessor systems (SMP) are composed of multiple identical processors and a single shared main memory. These

[1] Picture from www.anandtech.com

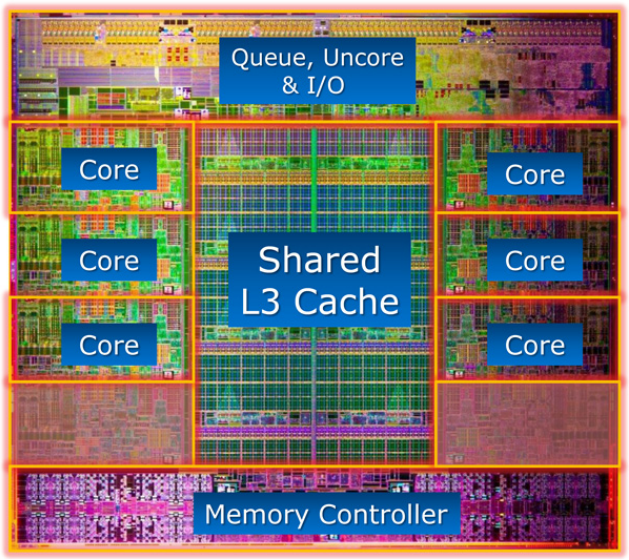


Figure 1.5: A Sandy bridge E die[1]

architectures are also referred to as Uniform Memory Access (UMA) architectures. The number of processors in a SMP system is limited, and the memory access is serialized, creating concurrency over the memory bus. The cache coherency mechanisms also send signals on the bus, which increases the traffic. At some point, the memory bus congestion becomes an issue for the performance [45]. A solution to this problem is the use of distributed memory clusters. However distributed memory programming creates constraints for the developer who has to manage data transfers explicitly. Also the granularity of the memory distributed between the different nodes degrades the performance.

Another solution was developed during the 90s to overcome the scalability limits of SMPs: Non Uniform Memory Access (NUMA) or cache coherent NUMA (ccNUMA) architectures as most of today's systems maintain cache coherency. CcNUMA systems are generally composed of several multicore processors and their memory banks (considered as multiple SMPs used as NUMA nodes). Each processor core is able to access any memory part. The cost of the memory access will be different, depending on the location of the data requested, but the whole memory is shared seamlessly for the developer, like for SMP architectures. This is supported by the Operating System (OS) which provides a virtual address space to the program. Figure 1.6 shows the topology of a NUMA system composed of two 6 core multiprocessors and their dedicated memory, forming two NUMA nodes.

CcNUMA architectures offer a better scalability than SMP systems and do not require the user to use message passing tools to explicitly distribute the data as it would be the case when using clusters. However, to achieve good scalability, parallel programs on ccNUMA systems should make good use of the cache memory to minimize memory access and ensure a good data locality (the data computed by

a thread on a core are in a local bank of memory inside the same NUMA node) in order to avoid remote access [43].

Performing efficiently dense linear algebra computations on ccNUMA systems requires to take into account the locality of the data and the memory access pattern of the algorithms. In Chapter 4, we will introduce some methods to efficiently use NUMA architectures on top of the dense linear algebra library MAGMA.

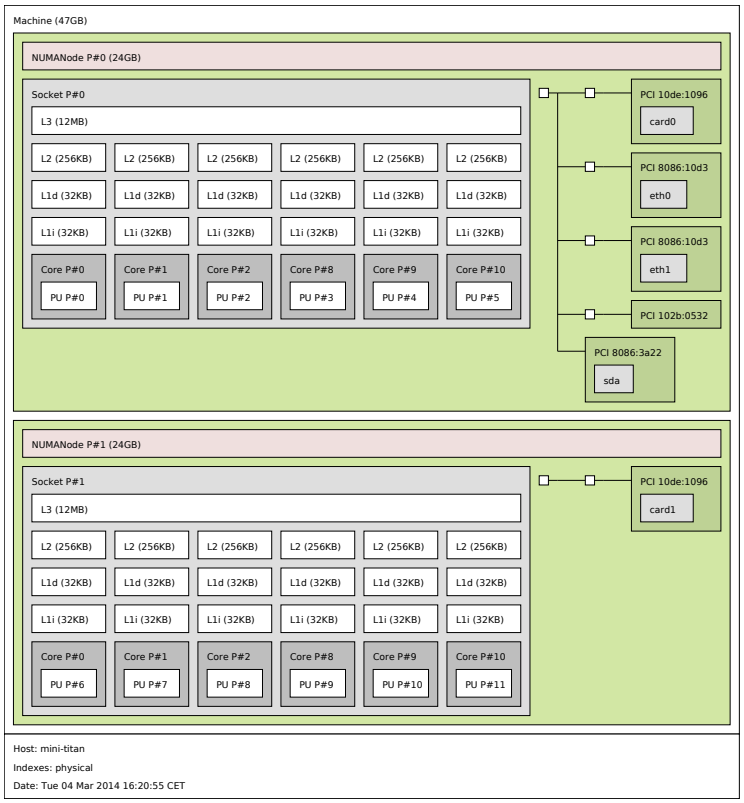


Figure 1.6: Topology of a computer with 2 NUMA nodes.

1.4.5 General Purpose Computation on Graphics Processing Units (GPGPU)

The graphics processing units (GPU) are specialized electronic circuits designed to create or accelerate the generation of images to be displayed. Before the 2000's, GPUs were only used to compute or accelerate the computation of 2D and 3D pictures. In 2001, Larsen developed one of the first example of non-graphical computation on a GPU with a matrix-matrix product [46] using the 8-bits integer texture maps. In 2003, the introduction of 32-bits floating-point values allowed not only a real progress in graphical processing but also in matrix computations on GPUs [47]. In 2005, Galoppo and al. developed an efficient LU factorization on GPU outperforming the optimized CPU implementations at that time. This was performed

using the languages and methods initially designed for graphics processing. In 2007, Nvidia released the Compute Unified Device Architecture (CUDA) programming platform [48] providing a virtual instruction set, allowing the development of general purpose applications without using the tools and languages designed for graphics processing only.

There are other GPGPU solutions such as DirectCompute [49], specific to Microsoft Windows, released in 2008 by Microsoft as a part of DirectX 11, and OpenCL [50] a framework maintained by the Khronos Group[2] consortium. OpenCL is designed to program parallel heterogeneous systems mostly CPUs and GPUs. It has also the advantage to work on GPUs other than Nvidia's ones (e.g., ATI).

Figure 1.7: Nvidia Tesla K40 GPU [3]

GPGPU has become a common occurrence in HPC and is often used in supercomputer architectures. GPGPUs offer a big computational capacity at a low cost and a good energetic efficiency. Out of the ten most powerful supercomputers in the latest TOP500 [51] ranking (November 2014), three use GPGPU accelerators. Today's Nvidia Tesla GPGPU Kepler K40 (showed in Figure 1.7) offers a theoretical performance of 4290 Gflop/s in single precision and 1430 Gflop/s in double precision.

The drawback of GPGPUs comes from their hybrid programming model that does not allow as much efficiency as CPU-only architectures due to its SIMD-only nature and the PCI-Express bandwidth limitations. As an example, in the LINPACK benchmark [52], the Titan supercomputer[4] using hybrid CPUs/GPUs architecture achieves 17590 Tflop/s out of 27112.5 Tflop/s, the theoretical peak performance corresponding to an efficiency close to 65%. While the Sequoia supercomputer[5]

[2] www.khronos.org
[3] Picture from www.nvidia.com
[4] www.olcf.ornl.gov/titan/
[5] http://computation.llnl.gov/computers/sequoia

(using only CPUs) achieves 17173.2 Tflop/s out of 20132.7 Tflop/s (around 85% efficiency).

Their highly parallel architecture model makes GPGPUs a suitable solution for matrix computations and dense linear algebra programs. However, programming efficiently on GPU-based architectures is a critical challenge for high performance computing. In this thesis we showed some solutions to efficiently solve dense linear systems, using GPGPUs as accelerators.

1.4.6 Intel Xeon Phi accelerators

In 2010 Intel announced their Many Integrated Core Architecture (Intel MIC) a highly parallel coprocessor architecture consisting of several x86 processor-cores and it's own GDDR5 integrated memory. The first prototype board called *Knights Ferry* consisted of 32 cores with 2GB of GDDR5 memory. In 2012 Intel released the *Knights Corner* product line branded as "Xeon Phi" [53].

The current Xeon Phi 7120 (like the ones in Figure 1.8) posses 61 cores with four threads per core, running at 1.238 GHz. It has 16 GB of GDDR5 memory on 16 channels for a maximum bandwidth of 352 GB/s. Each core has 512 KB of Level 2 cache memory for a total of 30.5 MB of cache memory. The cornerstone of the Xeon Phi performance is the Vector Processing Units (VPU) of each core, using 512 bits wide SIMD registers allowing to perform 16 single-precision (SP) or 8 double-precision (DP) operations per cycle. The SIMD instruction set also includes Fused Multiply-Add (FMA) allowing to perform 32 SP or 16 DP operations per cycle [54].

Using the FMA instructions, the peak performance of the Xeon Phi 7120 can be computed as:

$$\text{Clock Frequency} \times \text{Number of cores} \times \text{size of the lanes} \times 2(\text{FMA}) \text{ Flops}$$

We have then 1064.8 Gflop/s in DP and 2129.6 Gflop/s in SP [55].

In November 2014, two out of the ten most powerful supercomputers were using Intel Xeon Phi accelerators, including the number 1 of the TOP500 ranking, Tianhe-2 that uses 48000 Xeon Phi coprocessors and achieves a performance of 33862.7 Tflop/s out of a theoretical peak performance of 54902.4 Tflop/s (around 62% efficiency).

Multiple tools can be used to program the Intel Xeon Phi with efficiency: OpenMP can be used to lever the parallelism between threads, the Intel compiler can perform some simple auto vectorizations, and preprocessor directives can be used to handle the hybrid computing issues such as the memory transfers etc. Nevertheless, for advanced programs with non trivial parallelism, achieving high performance with the Xeon Phi can be challenging. In practical cases, advanced low level optimizations such as hand-written SIMD code are required [56].

The combination of wide SIMD registers and a high level of shared memory core-based parallelism allows the Intel Xeon Phi to perform dense linear algebra computation with a high level of performance. To achieve such a level of performance the implementation must take advantage of these architectural features. Like for

the GPGPU, in this thesis we provide an efficient dense linear solver using the Intel Xeon Phi as an accelerator.

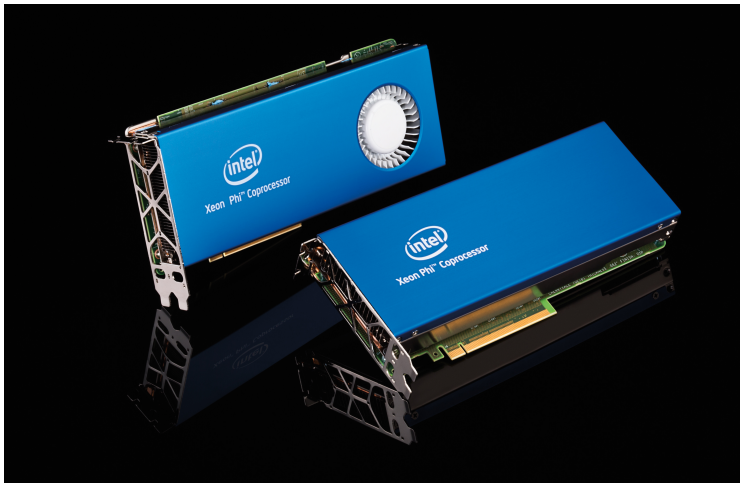


Figure 1.8: Intel Xeon Phi Coprocessors[6]

[6]Picture from www.intel.com

1.5 Numerical linear algebra libraries for dense matrices

The evolution of computer architectures has been followed by the software libraries. A different architecture requires a different implementation to be able to perform at the best of the capabilities of the machine. In dense linear algebra, the computational efficiency is a major challenge. Therefore an optimized implementation of the programs is essential.

Before the 70s, the optimized computational routines were directly coded in machine code. In 1964, Sweeney [57] collected statistics about the floating-point operations in various programs, in order to determine what kind of operations were mostly used [31].

The most usual operations were then included in black box libraries, offering the programmer optimized functions for different tasks such as matrix and vector operations (copies, swaps, rotations, etc).

1.5.1 The historical libraries

In the mid 60s, IBM distributed the Scientific Subroutine Package [58], a collection of FORTRAN Subroutines optimized for the IBM System/360 machine. In 1974, Garbow published EISPACK [59], a package of FORTRAN subroutines to compute eigenvalues and eigenvectors of matrices. The Basic Linear Algebra Subprograms (BLAS) package was first the result of a collaborative project of the ACM-SIGNUM committee carried between 1973 and 1977 [60]. Based on a proposal made in 1973 [61].

The LINPACK library proposed, in 1979 a set of subroutines designed for the supercomputers of the 70s and 80s, mostly based on vector processors, to solve linear equations and linear least-square problems [62]. LINPACK used BLAS for the basic matrix manipulations. The LINPACK user's manual included a benchmark. It used a LU factorization with partial pivoting to solve a problem of size 100, allowing users to estimate the performance of their memory and processors. This size of the test and its implementation has evolved but is still in use today. It is known as High Performance LINPACK (HPL) [52, 63] and allows to establish the ranking of the most powerfull supercompters referred to as the TOP500 [51].

The first version of BLAS (Level 1 BLAS) implemented scalar-vector and vector-vector operations. BLAS2 (Level 2 BLAS) was developed in 1988 as an extension to BLAS1 to take advantage of the capabilities of vector processors [64, 65]. BLAS2 allows to perform matrix-vector operations.

In 1990 BLAS3 [66, 67] added another extension to be "cache friendly". This extention takes into account the memory hierarchy of the new computers (global memory, cache memories, vector registers etc). It implemented matrix-matrix operations such as matrix products or solving triangular systems of equations.

Released in February 1992, LAPACK [68] supersedes LINPACK and EISPACK and achieves better performance. LAPACK focuses on solving: systems of linear equations, linear least squares problems, eigenvalue problems and singular value

problems. To perform these operations it also implements the associated computation such as matrix factorizations (LU, QR, LDLT, Cholesky etc) or the estimation of condition numbers. LAPACK uses BLAS routines as much as possible to perform all the matrix, vector, scalar computation. Therefore the performance of the LAPACK libraries depends on the implementation of the BLAS used.

Most of the numerical linear algebra libraries developed afterwards are based on BLAS and LAPACK, offering different implementations of the same routines and operations.

1.5.2 Parallel implementations

Some vendor libraries such as ACML [69] for AMD processors, MKL [39] for Intel processors and ESSL [70] for IBM provide optimized implementations of BLAS and LAPACK for their processors. These optimizations include multithreaded (for the multicore processors) and vectorized (for the SIMD extensions) functions.

Open source projects also exist such as ATLAS [71], Goto BLAS [72] or OpenBlas [73]. ATLAS uses a tuning step during its installation to determine the best parameters for the kernels, with respect to the target architecture. GotoBLAS possesses a collection of hand written assembly kernels optimized for different architectures and uses vectorization and multithreading. OpenBLAS is based on GotoBLAS2 and proposes optimizations for more recent architectures since the development of GotoBLAS stopped.

For GPUs, NVIDIA proposes CuBLAS [74], an implementation of BLAS on top of the NVIDIA CUDA runtime, and EM Photonics, their CULA [75] solution as a CUDA implementation of LAPACK.

ScaLAPACK [76] (Scalable LAPACK) is an implementation of LAPACK for distributed architectures. It is based on different libraries. BLAS and LAPACK for the computation on each node, Basic Linear Algebra Communication Subprograms (BLACS) for the communication tasks. BLACS uses MPI for communication between the nodes. ScaLAPACK uses block-partitioned algorithms for the computation and two-dimensional block-cyclic distribution for the storage of the matrices.

Solutions using different programming approaches exist, such as the Formal Linear Algebra Methods Environment (FLAME) [77] offering a more user-friendly Application Program Interface (API) to represent the algorithms using algorithmic skeletons. The contributors also developed the BLAS counterpart BLIS [78] using the same approach.

The Parallel Linear Algebra Software for Multicore Architectures (PLASMA) is a software library designed to be efficient on homogeneous multicore processors and multi-socket systems of multicore processors. PLASMA [79] achieves a much greater efficiency than LAPACK but does not support band matrices and does not solve eigenvalue and singular value problems. It does not replace ScaLAPACK since it does not support distributed architectures.

PLASMA uses BLAS kernels for its internal computation, so an optimized BLAS implementation is required to achieve good performance. PLASMA implementation

is based on tiled algorithms. The idea is to divide the matrices into small enough square tiles so that a tile fits into the cache memory of one core. This method minimizes the number of cache misses, improving the performance. To make a tile to be stored in the cache memory efficiently, each tile has to occupy a contiguous memory region. We note that the storage used in PLASMA is different from LAPACK in which the matrices are stored column wise. Here a tile layout is used where each tile is continuously laid out in memory.

Another principle of PLASMA is related to the dynamic task scheduling. The scheduler called QUARK [80] (QUeueing And Runtime for Kernels) uses task graphs or Direct Acyclic Graphs (DAG), which are generated and explored at runtime.

1.5.3 The MAGMA Library

Similarly to LAPACK, MAGMA[7] [81, 82, 83], is being build as a community effort, incorporating the newest developments in hybrid algorithms and scheduling, and aiming at minimizing synchronizations and communication in these algorithms. The goal of these efforts is to redesign the dense linear algebra algorithms in LAPACK to fully exploit the power of current heterogeneous systems of multi/manycore CPUs and accelerators, and deliver the shortest possible time to an accurate solution within given energy constraints. Indeed, the algorithms included so far in MAGMA 1.6 manage to overcome bottlenecks associated with just multicore or GPUs, to significantly outperform corresponding packages for any of these components taken separately. MAGMA's one-sided factorizations for example on a single Fermi GPU (and a basic CPU host) can outperform state-of-the-art CPU libraries on high-end multi-socket, multicore nodes (e.g., using up to 48 modern cores). The MAGMA library exists in three versions: one for Nvidia GPUs using CUDA, one using OpenCL and one dedicated to Intel Xeon Phi accelerators.

More details about the MAGMA library will be given in Chapters 2 and 3. In this thesis, we used MAGMA as a framework to develop new solvers. Some of these solvers have been included in the latest release of the GPU and Intel Xeon Phi versions of MAGMA.

1.6 Conclusion of Chapter 1

In this chapter, we discussed the different methods for solving dense linear systems of equations, and focused on the dense solvers based on the LU factorization. We addressed the issue of pivoting in Gaussian elimination and described existing pivoting strategies to improve the stability of the LU algorithm.

We also discussed the architectures used in high performance computing and their challenging exploitation for the programmer, due to the different types of parallelism and programming paradigms.

[7] Matrix Algebra on GPU and Multicore Architectures, `http://icl.cs.utk.edu/magma/`

We described the evolution of the numerical libraries used to solve dense linear systems, and showed the adaptations performed to offer the best performance as possible, depending on the targeted architectures.

In the next chapter we present our contributions in designing and implementing different algorithms to perform LU factorization on hybrid architectures using CPUs and GPUs. We also describe the behavior of the resulting routines in terms of performance and accuracy.

Chapter 2

Hybrid CPU/GPU algorithms for LU factorization

Contents

2.1 Dense linear algebra on accelerated multicore machines

There has been several main changes in the development of dense linear algebra libraries over the years. These changes have always been triggered by major hardware developments. For example, LINPACK [62] in the 70's targeted the vector machines at the time for which cache reuse was not essential, and as a result LINPACK had relied on just Level 1 BLAS. In the 80's LINPACK had to be rewritten, leading to LAPACK [68], that would rely on Level 3 BLAS for cache based machines. In the 90's it was extended to ScaLAPACK [76] for parallel platforms, relying on the PBLAS [84] message passing. Now, in the 00's, with the explosion in parallelism and heterogeneity as well as the ever increasing data-communication costs, the old libraries had to be redesigned once again. An example of these new generation libraries is the MAGMA library [85, 86, 82] (see Section 1.5.3) that has been designed from 2008 to address heterogeneous parallel architectures based on accelerators.

In parallel to the development of hybrid algorithms, there has been a number of new developments related to minimizing communication in one-sided factorizations

(e.g. [87]). Such improvements have become essential due to the increasing gap between communication and computation costs.

For the linear system solvers on current multicore or GPU architectures, a bottleneck in terms of communication cost and parallelism comes from the pivoting, a technique used to prevent divisions by too-small numbers in the Gaussian Elimination (GE) process (see Sections 1.3.2 and 1.3.3). The commonly used method of Gaussian Elimination with partial pivoting (GEPP) is implemented in current linear algebra libraries for solving square linear systems $Ax = b$ resulting in very stable algorithms (see 2). These systems are in general solved using the well-known LU factorization that decomposes the input matrix A into the product $L \times U$, where L is a lower triangular matrix and U is an upper triangular matrix. Current libraries like LAPACK implement GE using a block algorithm, which factors the input matrix by iterating over its blocks of columns (panels), as described in Section 1.3.5. Pivoting not only requires communication (or synchronization in a shared memory environment), but it also limits the exploitation of asynchronicity between block operations. This is because the update of the trailing submatrix can be performed only when the panel factorization is completed. We can find in [30] an evaluation of communication overhead due to partial pivoting using MAGMA on a given CPU/GPU architecture. This cost can represent on some hybrid architectures up to 40% of the global factorization time, depending on the matrix size. Communication cost of GEPP is asymptotically larger than the lower bounds on communication [24]. Other classical pivoting strategies can be used (see Section 1.3.2), but they always require between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ comparisons to search for the pivot. In this chapter, we consider two alternative strategies to these pivoting techniques, that have the property of reducing communication in the LU factorization while providing a satisfactory accuracy.

The first alternative, already described in Section 1.3.6.1, is tournament pivoting. It was introduced in the context of CALU, a communication-avoiding LU factorization algorithm [26]. It has been shown in [24] that tournament pivoting is as stable as partial pivoting in practice and that CALU minimizes communications. With this strategy, the panel factorization, referred to as TSLU (Tall and Skinny LU), can be efficiently parallelized.

The second alternative is proposed in [30] where the communication overhead due to pivoting is completely removed by considering a randomization technique referred to as Random Butterfly Transformation (RBT) (see Section 1.3.7). More details on this method will be given in Chapter 3.

Note that, since in this approach we know in advance that we are not going to pivot, GENP that follows randomization is implemented as a very efficient fully BLAS 3 algorithm. Note also that when the initial matrix is randomized, we systematically add iterative refinement in the working precision for better stability, as indicated in [31, p. 232]. We show in this chapter that the usage of these techniques in the context of hybrid CPU/GPU architectures lets us to take advantage of each computational unit.

The chapter is organized as follows. First, we describe in Section 2.2 how the

MAGMA library implements the LU algorithm with partial pivoting on hybrid architectures with one or multiple GPUs. We also adapted this method to implement the LU factorization with no pivoting used for the RBT solver.

Then, in Section 2.3 we introduce tournament pivoting, a strategy based on CALU that we adapted specifically for CPU/GPU architectures. In this new implementation, the panel is factored on the CPU using a modified CALU factorization while the update of the trailing submatrix is performed on the GPU. The resulting solver is called H-CALU solver. Finally we propose in Section 2.4.1 some performance results for the panel and the whole matrix factorizations where we compare H-CALU with LU solvers using partial pivoting (MAGMA) and RBT. Concluding remarks are given in Section 2.5.

2.2 MAGMA implementations of LU factorization

2.2.1 Mono GPU implementation

Let us illustrate how the hybrid multicore + GPU approach can be applied to the LU factorization by describing the algorithm as it is implemented in the MAGMA library. The method is based on splitting the computation as shown in Figure 2.1 that represents a current matrix factored via a right looking block LU factorization [1, p. 85], where the dark part has been already factored. The initial matrix has been downloaded to the GPU and we describe in algorithm 3 a current iteration:

Algorithm 3 Iteration for LU factorization using MAGMA

1: The current panel (1) is downloaded to the CPU.
2: (1) is factored by the CPU using GEPP and the result is sent back to the GPU.
3: The GPU updates (2) (next panel).
4: The updated panel (2) is sent back to the CPU to be factored while the GPU updates the rest of the matrix (3).

The technique consisting of factoring (2) while still updating (3) is often referred to as *look-ahead* [88]. In the current implementation of MAGMA, the panel factorization is performed using GEPP but this algorithm is general enough to be applicable to many forms of LU factorizations, where the distinction can be made based on the form of pivoting that they employ. In Section 2.3 we use a different pivoting strategy that turns out to be very efficient for factoring the panel due to its particular "tall and skinny" structure. Depending on the problem size n and on the hardware used, MAGMA proposes a default value for the parameter b (width of the panel).

Note that the design of the hybrid LU in MAGMA avoids communicating by having only panels transferred between CPU and GPU ($\mathcal{O}(n * b)$ data vs $\mathcal{O}(n * n * b)$ computation in the updates), enabling also the total overlap of the panel computation by the updates for n large enough.

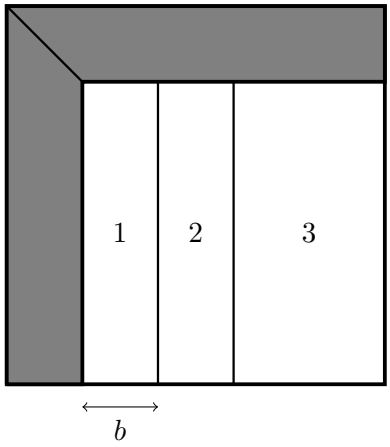


Figure 2.1: Block splitting in hybrid LU factorization

2.2.2 Multi GPUs implementation

Algorithm 4 Magma LU multi GPU without pivoting

Input: id is the identifier of the GPU.
Input: num_gpus is the number of gpu used.
Input: The matrix is distributed on the GPUs using a 1-D cyclic block column layout.
Input: M is the size of the matrix and nb the panel size.

1: $steps \leftarrow M/nb$
2: **for** $i \leftarrow 0$ to $steps - 1$ **do**
3: $panel\_owner \leftarrow i \mod num\_gpu$
4: **if** $panel\_owner = id$ **then**
5: asynchroniously send the panel to the CPU.
6: Synchronize to ensure that the task queue is empty.
7: **end if**
8: **if** $i > 0$ AND $panel\_owner = id$ **then**
9: Update of its trailing submatrix.
10: **end if**
11: **if** $panel\_owner = id$ **then**
12: Barrier on the previous panel sending.
13: **end if**
14: CPU factorize the panel.
15: CPU asynchroniously send the factorize panel to all the GPUs.
16: **if** $panel\_owner = id$ **then**
17: Storing the factorized panel in the place from where it was taken.
18: **else**
19: Storing the factorized panel in temporary matrix.
20: **end if**
21: Barrier on the panel reception.
22: **if** $id = (panel\_owner + 1 \mod num\_gpus)$ **then**
23: Update the trailing submatrix corresponding to the next panel. {Look-ahead}
24: **else**
25: Update its trailing submatrix.
26: **end if**
27: **end for**

Algorithm 4 presents a no pivoting version of the factorization that can be used in the RBT solver and uses multiple GPUs. The matrix to factorize is distributed on the GPUs as shown in Figure 2.2 using a 1-D block-cyclic column layout [76, p. 58]). At each step the current panel is downloaded from the GPU that owns it to the CPU to be factored. When the CPU finishes the panel factorization, it sends it to all GPUs. This panel is stored in a temporary space allocated on each

GPU (except for the GPU that owns this panel from the data distribution) and the GPUs update their trailing submatrix. The GPU that owns the next panel updates in priority the part of the trailing submatrix that corresponds to the next panel and sends it to the CPU. Using this algorithm, we can compare in Figure 2.10 the performance of the LU with partial pivoting and no pivoting. It shows that using multiple GPUs is interesting only when we consider large systems since for smaller sizes, communication cost between CPU and GPUs is significant. Note also that the no-pivoting factorization is much more scalable than the partial pivoting factorization. Indeed, the latter does not take full advantage of the multiple GPUs since the pivoting is performed on the CPU. This justifies again the interest of using techniques to avoid pivoting on these architectures. This aspect will be developed in Chapter 3.

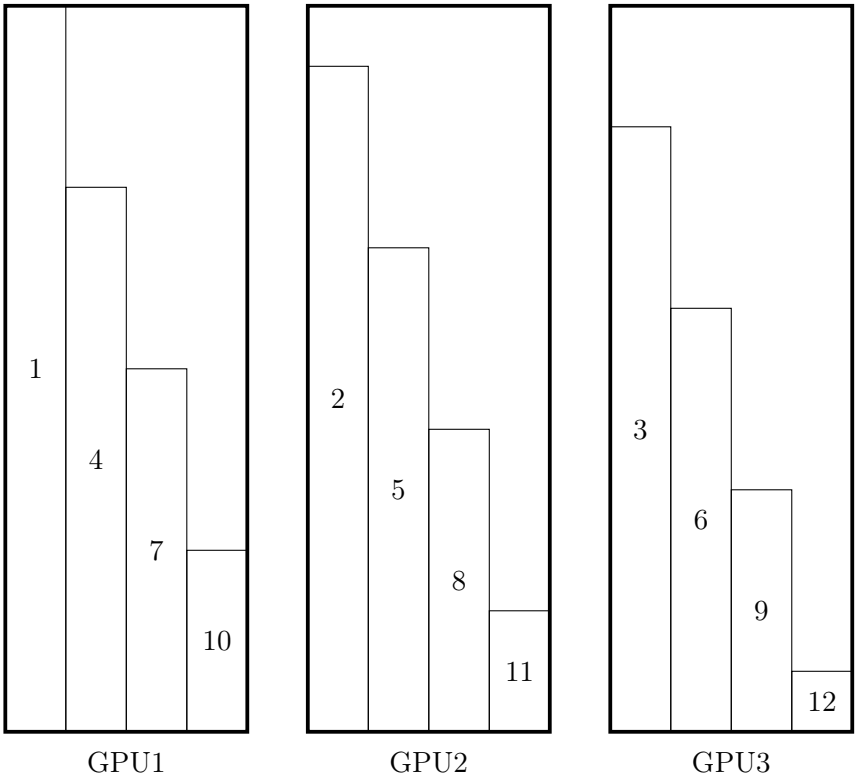


Figure 2.2: Example of orders for the panels pactorizations in Magma with 3 GPUs and a panel size being 1/12 of the matrix size.

2.3 Hybrid implementation of tournament pivoting LU

The poor evolution of latency and memory bandwidth that we observed over recent years for parallel architectures is a major bottleneck for algorithms that require communication like GEPP. New approaches have been recently proposed to minimize

the amount of communication due to pivoting. Among them are communication-avoiding algorithms introduced for distributed-memory machines [26] and for multi-core architectures [25]. These algorithms are effective in the sense that they reduce significantly communication while being stable in practice. CALU is an algorithm that partitions the input matrix into block of columns (panels), iteratively factors the panel and updates the trailing submatrix. The factorization of the panel is one of the most important task in the LU fatorization since it is part of the critical path in the diagram of tasks and its effective execution influences the performance of the algorithm. In CALU, the panel factorization is performed by the TSLU algorithm which factors a block column of size b (see 1.3.6.1).

Once the panel is factored using TSLU then we update the trailing submatrix. Following the approach presented in [89, 90], the CALU algorithm can be represented as a Directed Acyclic Graph (DAG) where nodes are elementary tasks that operate on one or several $b \times b$ blocks and where edges represent the dependencies among them. A dependency occurs when a task must access data that is the output of another task, either to further update or just read that data. In Figure 2.3 we represent an example of LU factorization with CALU as a sequence of DAGs using 2 threads. The panel is partioned into 3 column blocks. Black tasks represent the factorization of the panel via TSLU and the gray tasks represent the update of the trailing submatrix.

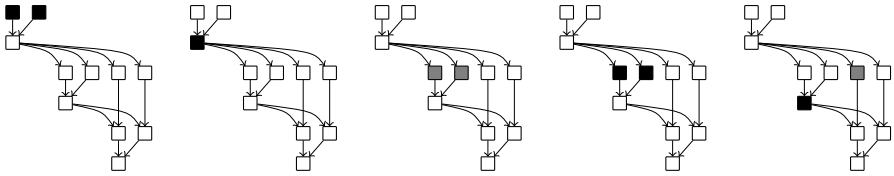


Figure 2.3: Example of asynchronous LU factorization using multithreaded CALU (2 threads, 3 column blocks) on CPUs

As explained in Section 2.2, the LU algorithm implemented in MAGMA factors each block of columns iteratively. Each step is essentially decomposed into two distinct phases: the factorization of the panel by the CPU followed by the update of the trailing submatrix by the GPU. The algorithm's design minimizes CPU-GPU communications. In the following we describe a method that further improves the algorithm in MAGMA by minimizing communication associated with the panel factorizations.

At each step, a panel of size B is factored on the CPU by applying CALU to a rectangular matrix and the update of the trailing submatrix is performed by the GPU. CALU factors the panel by splitting the initial block of columns into smaller blocks containing b columns that are factored iteratively using TSLU. Thus, the factorization of the panel is considered as a variant of the algorithm at the first level where we factor a rectangular matrix using only the CPU. The use of this second level of blocking is important for performance on hybrid CPU/GPU architectures

because the CPU and GPU processors have different size of cache. The block size B is chosen in order to optimize the performance of the matrix-matrix product on the GPU and to ensure a good grain for increasing parallelism. Then the block size b is tuned in order to optimize the utilization of the multicore cache. This decomposition of the algorithm into small tasks allows us to operate on blocks of data that fit into the cache. It results in an asynchronous and dynamic execution of the panel factorization on the CPU, yielding good performance on multicore machines [25]. This asynchronous execution keeps busy most of the CPU threads. When $b = B$, CALU behaves simply as TSLU. If B is large enough (which will be the case for our hybrid implementation), the panel is factored using CALU rather than TSLU because CALU can be executed asynchronously [25]. Our approach also uses the well known technique referred to as *look-ahead* [88] but adapted here so that the CPU and the GPU can work together while minimizing the number of memory transfers. In this approach, we start factoring the next panel as soon as possible.

Figure 2.4 depicts an example of the factorization of a matrix. We consider that the matrix is initially stored on the GPU. Black tasks represent the factorization of the panel using multithreaded CALU and the gray tasks represent the update of the trailing submatrix in the GPU. At each step of the factorization, the block corresponding to the panel is transfered to the CPU and factored using CALU. Once the panel is factored, it is sent back to the GPU in order to update the trailing submatrix. The GPU updates in priority the column block corresponding to the next panel. Note that, similarly to [82], the data transfer between CPU and GPU is overlapped by computation.

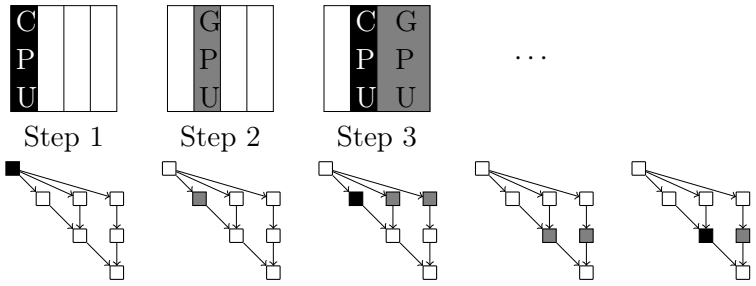


Figure 2.4: Hybrid CALU factorization (4 panels).

2.4 Performance comparisons

2.4.1 Experimental framework

In this section we present performance results for the algorithms described in Sections 2.2 and 2.3. These numerical experiments were carried out using a hybrid CPU/GPU system where:

- The GPU device is an NVIDIA Fermi Tesla S2050 with 448 CUDA cores running at 1.15 GHz and 2687 MB memory.

- The multicore host is a 48 cores system (4 sockets $\times$ 12 cores) AMD Opteron 6172 (2.1 GHz).

For experiments involving only the multicore host (panel factorization), comparisons are made against the MKL [39] multithreaded library. For experiments involving both CPU and GPU (factorization of the whole matrix), comparisons are made against version 1.1 of the MAGMA library. All computations are performed on random matrices and in double precision arithmetic.

2.4.2 Performance for the panel factorization

As described in Section 2.3, the panel factorization is performed by the CPU while the update of the trailing submatrix is executed on the GPU. Let us evaluate specifically the performance for the panel factorization phase in an LU factorization. This performance is measured by summing the total number of flops executed in factoring successively each panel throughout the factorization and dividing it by the time spent during these steps. This performance (expressed in Gflop/s) is plotted in Figures 2.5, 2.6, 2.7 and 2.8 for the factorization of four square matrices, each associated with a given panel size (parameter B defined in Section 2.3, corresponding to the number of columns for the panel). For factoring the panel, we consider different number of threads (one CPU core being used for each thread) varying from 1 to 26. Note that using more than 26 threads does not provide us with better performance, due to the too-large amount of communication involved in the panel factorization. The panel size B considered in Figures 2.5, 2.6, 2.7 and 2.8 for each matrix size corresponds to a value empirically tuned in order to provide the best global factorization time for each matrix when using a hybrid implementation.

In these experiments, we compare the performance of the panel factorization for the following routines:

- CALU factorization routine modified for the H-CALU solver and linked with the sequential version of MKL for the required BLAS and LAPACK routines.

- MKL implementation of the LAPACK routine `dgetrf`, used in the MAGMA implementation of LU for factoring the panel.

- A recursive routine for GEPP `rgetf2` (linked with MKL multithreaded BLAS) described in [91] and known to give good performance on "tall and skinny" matrices.

- GENP routine `dgetrf_nopiv` (no pivoting) as used in the RBT solver.

The routines compared in this section have been selected on the fact that they can be used as kernels for our hybrid CPU/GPU implementation. If we use only

multicore machines without GPU, then other solvers can be considered (see e.g. recursive tile version in [92]).

For the MKL implementation (partial pivoting LU, `dgetrf` routine), the best performance is achieved with 9 threads for sizes 5120 (10.77 Gflop/s) and 10240 (9.65 Gflop/s) and with 6 threads for size 15360 (12.82 Gflop/s) and 21504 (14.82 Gflop/s).

For the recursive LU factorization (partial pivoting LU, `rgetf2` routine linked with MKL multithreaded BLAS), the best performance is achieved with 15 threads for size 5120 (9.37 with 17 threads for size 10240 (12.02 Gflop/s), with 22 threads for size 15360 (18.02 Gflop/s), and with 24 threads for size 21504 (23.74 Gflop/s).

For CALU using sequential MKL kernels, the best performance is achieved with 12 threads for and with 16 threads for size 10240 (15.35 Gflop/s), for size 15360 (24.37 Gflop/s), and for size 21504 (30.66 Gflop/s).

The performance of the GENP routine can be considered here as a "peak" performance for the panel factorization. In this respect, we observe that, in percentage of this peak performance and depending on the matrix size n, CALU achieves between 36 % ($n = 5120$) and 48 % ($n = 21504$), `dgetrf` achieves between 35 % ($n = 5120$) and 23 % ($n = 21504$), and `rgetf2` achieves between 31 % ($n = 5120$) and 38 % ($n = 21504$). We also observe that CALU is even faster for larger ratios rows/columns. Moreover, CALU and GENP have better scalability properties. This can be explained by the fact that CALU increases the data locality thanks to its pivoting strategy and GENP does not pivot at all. The plateau observed for each curve after a certain number of threads corresponds to cases where the volume of communication becomes too large and cannot be overlapped by computation. For $n = 5120$, CALU, `dgetrf` and `rgetf2` give similar performance. However, when the matrix size increases and then the panel becomes more "tall and skinny", CALU outperforms the two other solvers and achieves a reasonable fraction of the GENP rate. This good behavior of CALU for factoring the panel was already mentioned in [25]. In particular this better scalability of CALU enables us to use more CPU threads in factoring the panel and then to improve the overall performance of a hybrid LU solver.

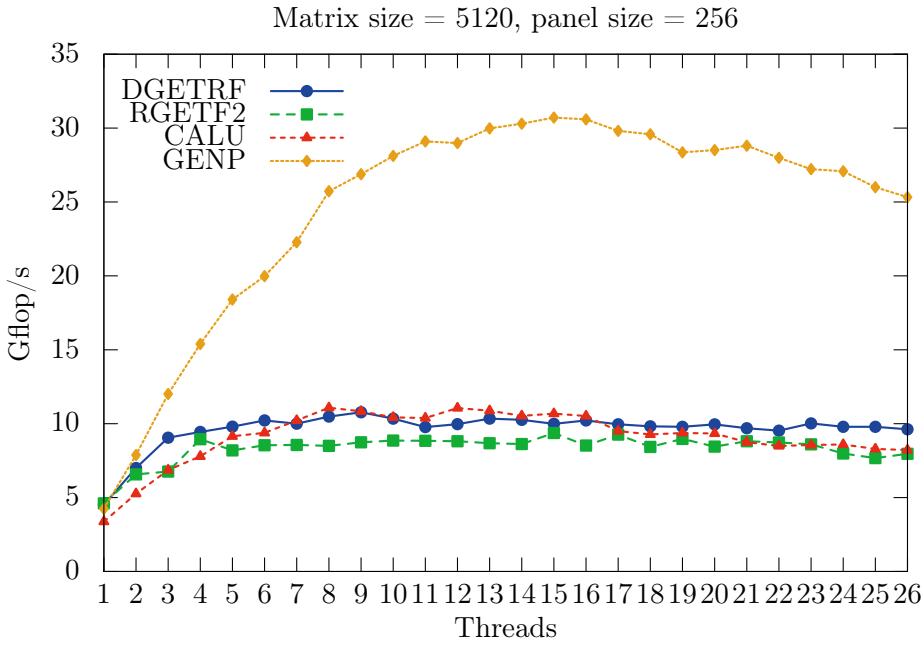


Figure 2.5: Comparison of CPU multi-threaded panel factorizations.

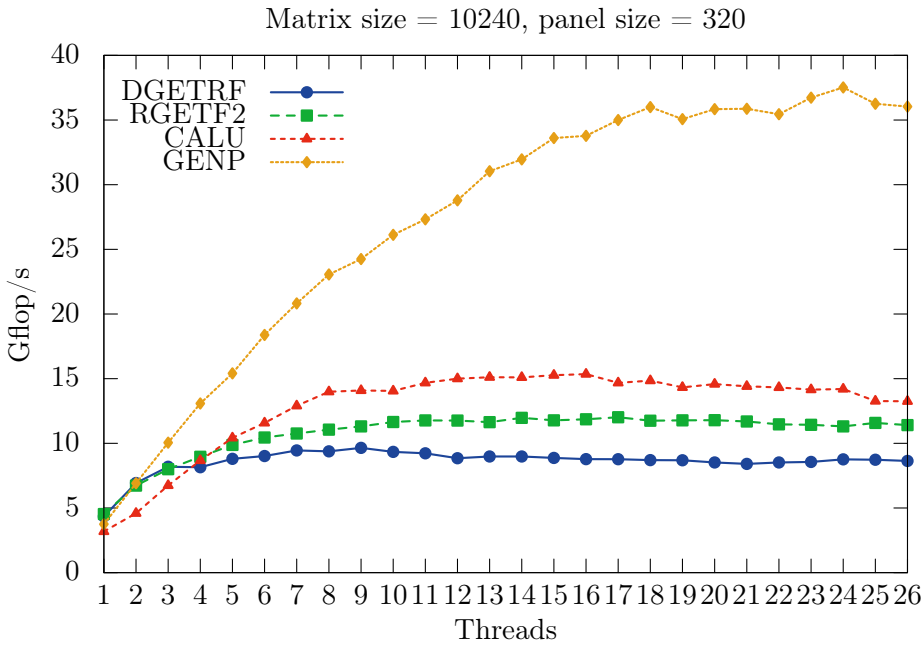


Figure 2.6: Comparison of CPU multi-threaded panel factorizations.

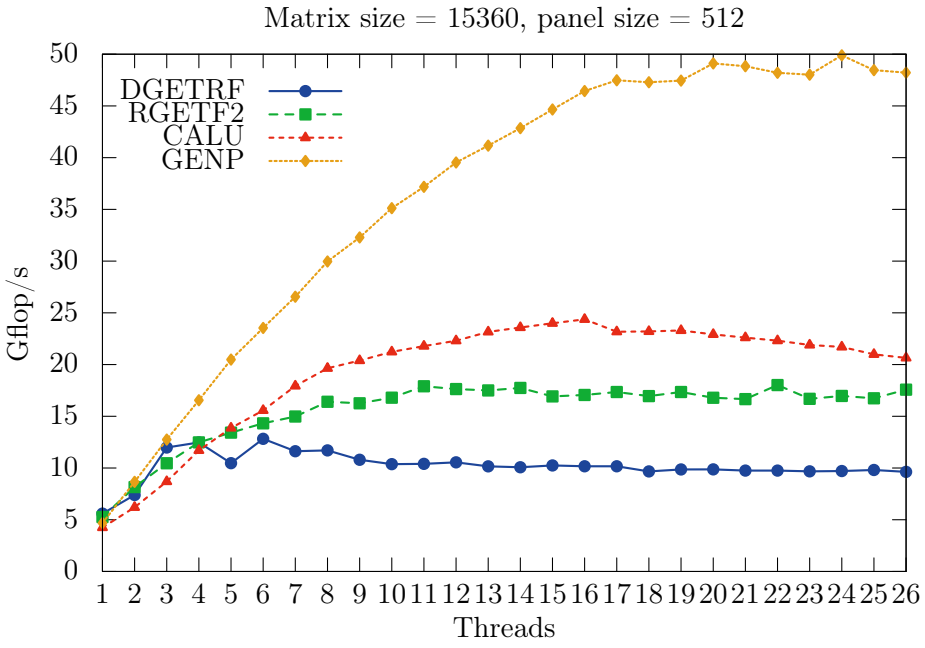


Figure 2.7: Comparison of CPU multi-threaded panel factorizations.

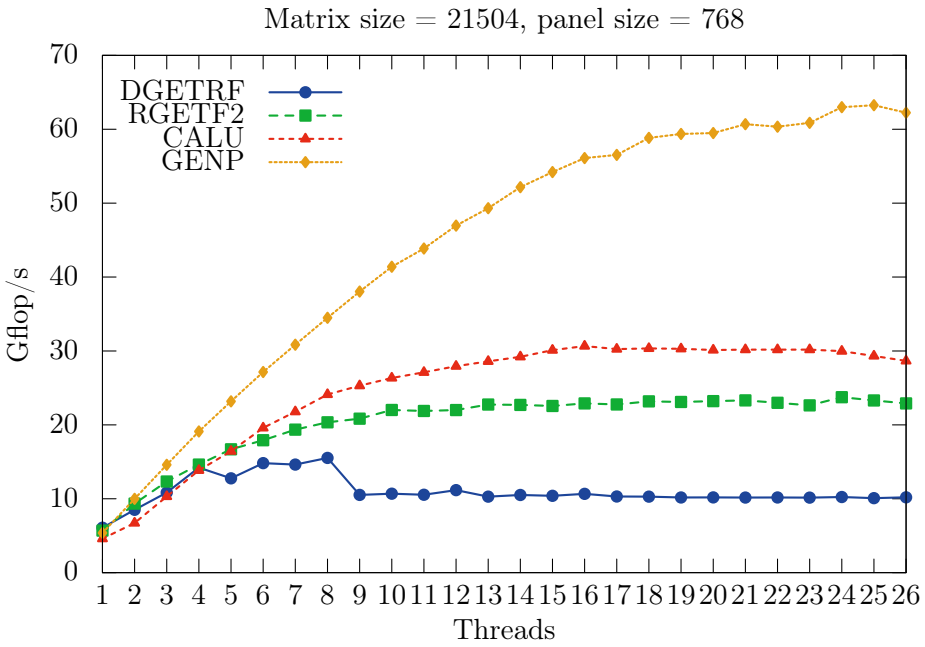


Figure 2.8: Comparison of CPU multi-threaded panel factorizations.

2.4.3 Performance for the hybrid LU implementations

In this section we study the performance of LU factorization routines that utilize resources from multicore (16 threads) and one GPU. We compare in Figure 2.9 the following routines, applied to square matrices of various sizes:

- The MAGMA routine `magma_dgetrf`, where the panel is factored using the MKL routine `dgetrf`,
- H-rgetf2, where the panel is factored with the recursive routine for GEPP `rgetf2`,
- H-CALU, where the panel is factored using the CALU routine mentioned in Section 2.4.2,
- The RBT solver (randomization + GENP).

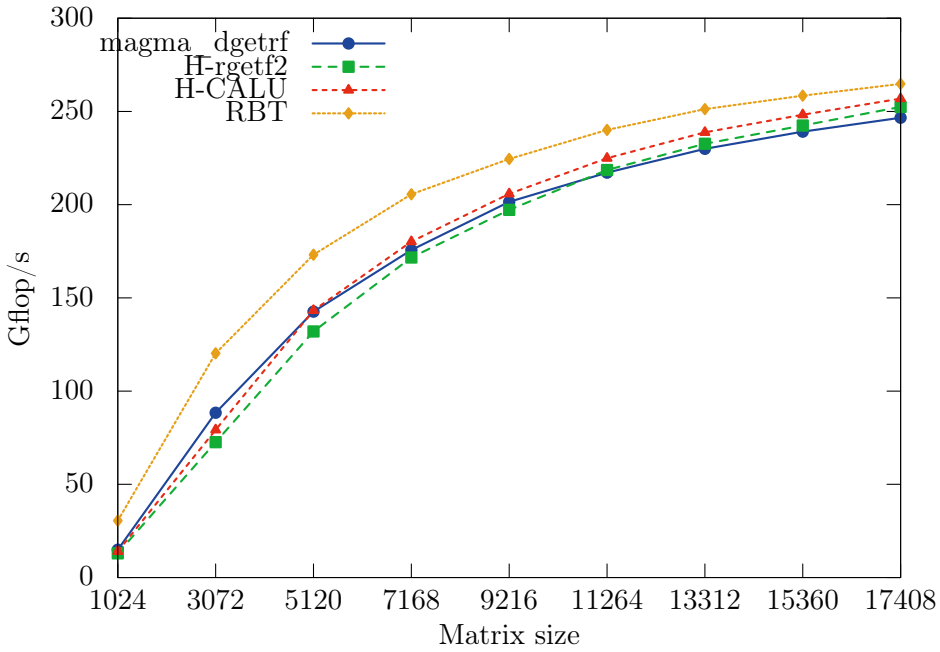


Figure 2.9: Performance on square matrices

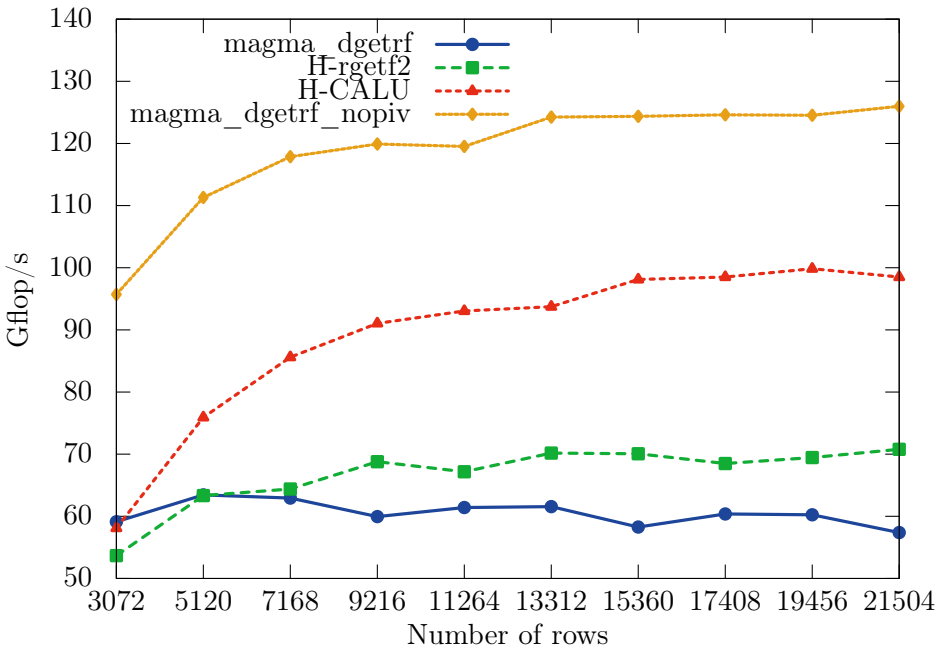


Figure 2.10: Performance on rectangular matrices

As expected, RBT outperforms the routines because it does not pivot and the randomization time is negligible. We can also observe that in the range 1024-5120, H-CALU gives similar performance as MAGMA but it is slightly faster for matrix sizes larger than 5120. This trend can be explained by the fact that, for matrix sizes smaller than 5120, the panels are not "tall and skinny" enough to take advantage of the CALU algorithm. We notice that the difference of performance observed for the panel in Section 2.4.2 has a moderate impact on the whole factorization since the update phase performed on the GPU represents the bulk of the computation. In these results the experiments performed with a MAGMA routine modified so that the panel is factored by the routine `rgetf2` mentioned in Section 2.4.2 obtained performance results similar to that of `magma_dgetrf`. Note that asymptotically, the performance of the three routines should be close because communication becomes negligible compared to the $O(n^3)$ computations for large dimensions.

In Figure 2.10 we compare the performance of hybrid LU factorization routines for rectangular matrices of size $m \times n$ with $m > n$, using 16 threads. Such an LU factorization exists when $A(1:k;1:k)$ is nonsingular for $k = 1:n$ (see [13, p. 102]). In our experiments $n = 2048$ and m varies from 3072 to 21504. Comparisons are made against MAGMA routines `magma_dgetrf` and `magma_dgetrf_nopiv` (instead of RBT since the latter has no implementation for rectangular matrices). We also compare with `H-rgetf2`, the MAGMA routine modified by factoring the panel using the recursive GEPP kernel.

On this type of matrices, H-CALU outperforms `magma_dgetrf` and H-`rgetf2`. Indeed, for rectangular matrices, the proportion of computation performed during

the panel factorization is bigger. Hybrid factorization on rectangular matrices could be for instance useful in a future hybrid factorization with multiple GPUs where the (rectangular) panel could be factored using CPU and a GPU.

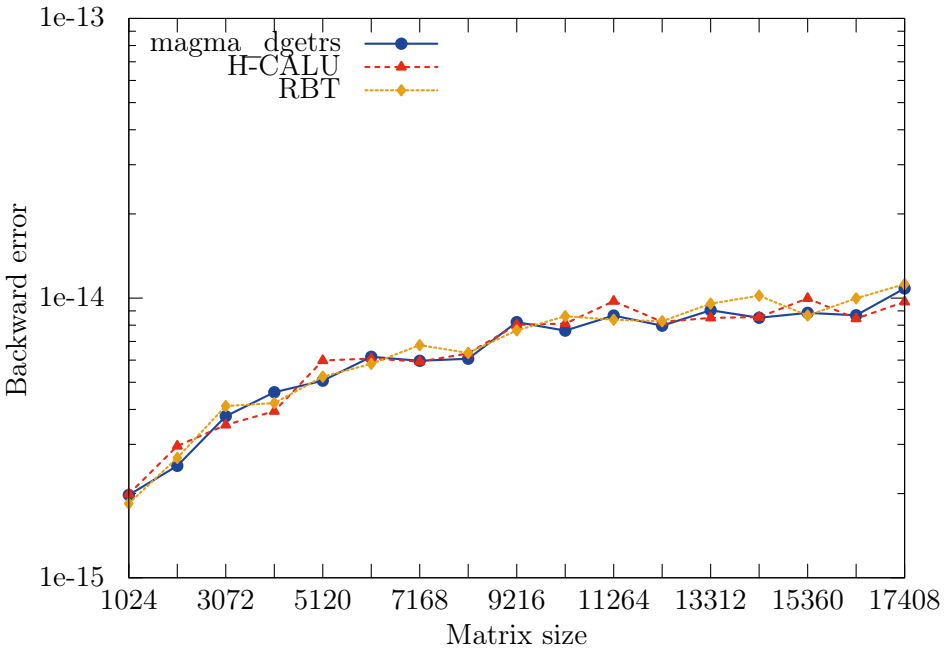


Figure 2.11: Comparison of componentwise backward error

In these experiments, we study the backward error obtained for the linear system solution computed with the solvers LU MAGMA, RBT and H-CALU, using the matrices considered in the previous experiments. The quantity plotted in Figure 2.11 corresponds to a componentwise backward error as defined in [68, p. 78]. Note that the number of threads used in factoring the panel in H-CALU is 16. This is mentioned here only because, as explained in [24], this might affect the accuracy. We observe that the backward errors are very similar for the three hybrid solvers. Tests on accuracy for specific matrix collections can be found in [30] and [24] respectively for RBT and CALU. Note that we did not perform accuracy tests using the routine `H-rgetf2` since, as it is based on partial pivoting, it would give the same numerical results as the `magma_dgetrs` routine.

Table 2.1: Test matrices

1	Diagonal	7	Last $n/2$ columns zero
2	Upper triangular	8	Random, $\kappa = \sqrt{0.1/\varepsilon}$
3	Lower triangular	9	Random, $\kappa = 0.1/\varepsilon$
4	Random, $\kappa = 2$	10	Scaled near underflow
5	First column zero	11	Scaled near overflow
6	Last column zero		

Table 2.2: Componentwise Backward Error

Matrix Type	MAGMA GEPP	h-CALU	RBT
1	0.0	0.0	2.10145e-16
2	1.31539e-16	1.31539e-16	2.18841e-16
3	184697e-16	184697e-16	2.06543e-16
4	2.16476-16	2.75832e-16	1.92510e-16
5	-	-	2.66472e-16
6	-	-	2.14281e-16
7	-	-	1.97144e-16
8	2.10408e-16	3.76095e-16	1.55625e-16
9	2.70036e-16	6.36540e-16	1.08967e-13
10	7.59577e-14	7.40225e-14	7.54745e-14
11	2.27295e-16	2.11000e-16	2.42990e-16

2.4.4 Performance on multiple GPUs

In this section we study the scalability using multiple GPUs for the partial pivoting routine from MAGMA (routine `magma_dgetrf_mgpu`) and a no pivoting factorization adapted from the MAGMA routine. In Figure 2.12, we observe that the no pivoting routine has a much better scalability. This difference in performance between both factorizations can be explained by the fact that the panel factorization and the swapping of rows due to partial pivoting are performance bottlenecks. By accelerating the panel factorization and removing the pivoting, we obtain a better scalability. However, it is important to note that the LU factorization with no pivoting is not stable when used alone in a linear system solver (unless the matrix is diagonally dominant) but it could be used in an RBT solver using multiple GPUs since the RBT preprocessing enables us to avoid pivoting.

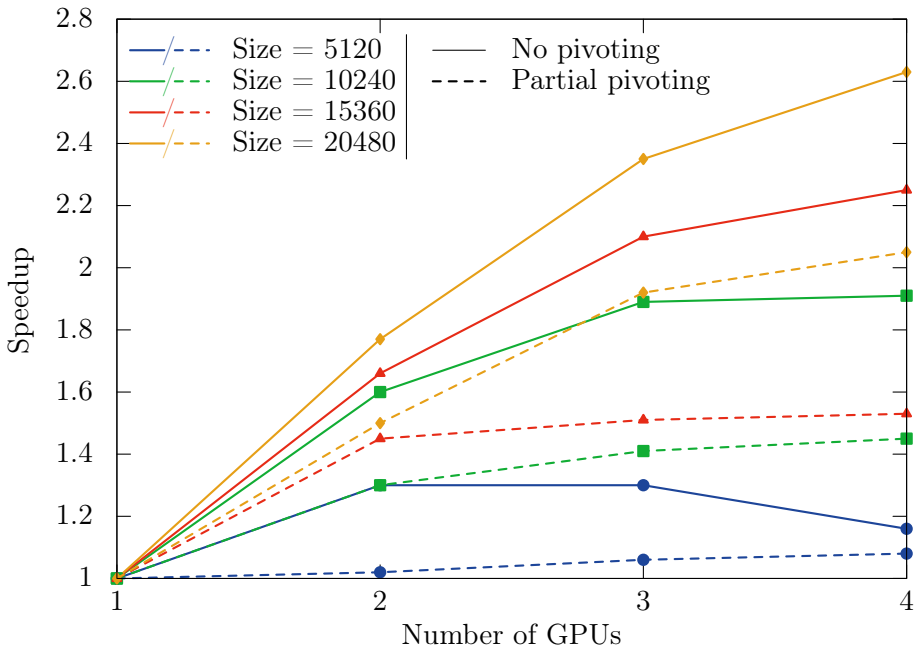


Figure 2.12: Performance of LU factorizations on multiple GPUs

2.5 Conclusion of Chapter 2

In this chapter, we presented different LU factorization routines using a multicore machine accelerated with one GPU and proposed a no pivoting version for multiple GPUs that can be used in the RBT solver. The difference between these approaches comes from the pivoting strategy chosen for factoring the panel. We proposed a new hybrid communication-avoiding solver H-CALU where the panel is factored on the CPU while the update is performed by the GPU. In our experiments, this solver turns out to be faster than the classical GEPP implementation in MAGMA for square matrices larger than 5120, and when using a sufficient number of threads. The good performance of H-CALU compared to partial pivoting allows us to consider larger panels in the block column factorization and thus to limit the amount of transfers between the CPU and the GPU memory. We point out that further optimizations are possible with e.g. additional tuning and scheduling, but our experiments give a general trend for the performance of algorithms as dictated by the amount of communication that they perform. However, the solver based on RBT always outperforms the other solvers since we do not pivot at all and the randomization cost is small. The methods and implementations of RBT-based solvers will be detailed in Chapter 3. The good performance of H-CALU on rectangular matrices is promising in the perspective of extending this approach to multiple GPUs. Part of the work described in this chapter has been published in [93].

Chapter 3

A fast randomized solver for accelerated multicore systems

Contents

3.1 Introduction

In order to solve general dense linear systems, the LU factorization with partial pivoting is the most commonly used method. However, even if pivoting ensures some numerical stability and does not require extra floating-point operations, searching for the pivot involves $\mathcal{O}(n^2)$ comparisons and swapping the rows of the matrix involves irregular data movements. These aspects can badly impact the performance due to cache invalidation it induces.

To avoid the cost of pivoting, and therefore improve the performance of the factorization, Random Butterfly Transformation (RBT) was proposed. This method, first described in [28, 29] was recently developed for general systems in [30] and for symmetric indefinite systems in [94, 95, 96]. Tests performed in [30] for a collection of test matrices showed that in practice two recursions are sufficient to obtain a satisfactory accuracy.

The RBT solvers are particularly suitable for accelerators. On the one hand, avoiding pivoting on accelerators has an important impact on performance, given that the rows do not need to be swapped. On the other hand, the structure of the butterfly matrices can be exploited to perform the randomization at a very

low cost. To use accelerators in dense linear algebra computations, we base our work on the MAGMA library, which provides LAPACK interface functions, using GPUs [81, 82, 83] or Intel Xeon Phi [97, 98].

This chapter is organized as follows. In Section 3.2, we describe the structure of the recursive butterfly matrices and how they can be stored in a compact format. We also describe the RBT algorithm and our implementation of the iterative refinement using MAGMA. In Section 3.3, we explain how the RBT solver can be performed efficiently on hybrid architectures using accelerators. In particular we detail how the randomization is applied. In Section 3.4, we present the implementation of the RBT solver included in the MAGMA library for GPUs using CUDA. We give details about the implementation of the corresponding GPU routines and we provide performance results in Section 3.4.2. In Section 3.5, we present our implementation of the RBT solver using Intel Xeon Phi coprocessors and give performance results in Section 3.5.2. Finally, we give some concluding remarks and ongoing work in Section 3.6.

3.2 RBT solver

As mentioned in Section 1.3.7, a butterfly matrix is an N-by-N matrix defined as follows:

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix},$$

Where R and S are two random non singular $N/2$-by-$N/2$ diagonal matrices.
The data pattern of a butterfly matrix can be represented as follows:

$$B = \left(\begin{array}{cc} \diagdown & \diagup \\ \diagdown & \diagup \end{array} \right),$$

We observe that the N-by-N butterfly matrices are only composed of two diagonals of size $N/2$. An N-by-N butterfly matrix can then be stored in an N elements vector with the $N/2$ first elements being the values of R and the $N/2$ last elements being the coefficients of S.

If we consider a recursive butterfly matrix of depth d as defined in [30], then it has the following recursive form:

$$W^{<n,d>} = \begin{pmatrix} B_1^{<n/2^{d-1}>} & & 0 \\ & \ddots & \\ 0 & & B_{2^{d-1}}^{<n/2^{d-1}>} \end{pmatrix} \times \cdots \times \begin{pmatrix} B_1^{<n/2>} & 0 \\ 0 & B_2^{<n/2>} \end{pmatrix} \times B^{<n>}, \tag{3.1}$$

where all $B_i^{<n>}$ blocks are size N butterfly matrices. We notice that all the matrices involved in the products can have their coefficients stored in N elements vectors. Thus the depth d recursive butterfly matrix can be stored in a d-by-N matrix as illustrated in Figure 3.1.

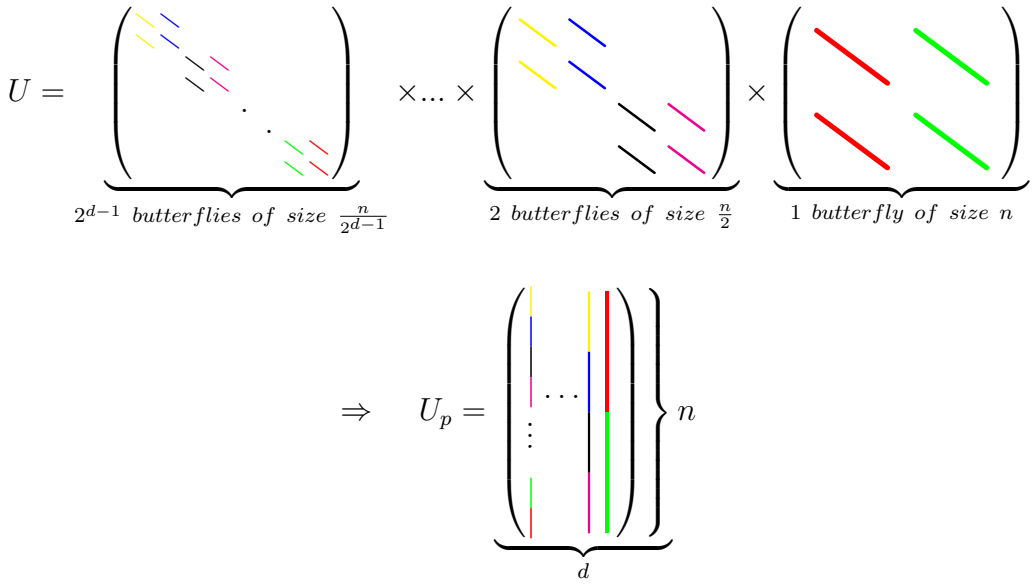


Figure 3.1: Packed storage for a recursive butterfly matrix

To solve the general linear system $Ax = b$ using RBT, we perform the following steps:

1. Compute the randomized matrix $A_r = U^T AV$, where U and V are recursive butterflies.

2. Factorize A_r using Gaussian Elimination with No Pivoting (GENP).

3. Solve $A_r y = U^T b$ (two triangular systems).

4. Use iterative refinement to improve the computed solution.

5. Solution is $x = Vy$.

As mentioned in Section 1.3.2, the GENP algorithm can be unstable due to a potentially large growth factor. This is why we systematically perform iterative refinement on the computed solution of the randomized system. Algorithm 5 describes how the iterative refinement is performed in our implementations. We have included this implementation in the MAGMA library. We improve the computed solution until we reach the required accuracy or we reach a defined maximum number of iterations (here 30).

Algorithm 5 Iterative refinement

Input: A the original matrix.
Input: b the right hand side.
Input: x the computed solution.
Input: L and U the factorized form of A
Input: N size of the matrix A
Result: An improved solution x

1: $\varepsilon \leftarrow$Machine precision
2: $IterMax \leftarrow 30$
3: $Iter \leftarrow 0$
4: $Anrm \leftarrow \|A\|_\infty$
5: $Xnrm \leftarrow \max|x|$
6: $Cte \leftarrow Arnm \times \varepsilon \times \sqrt{N}$
7: $r \leftarrow b - Ax$
8: $Rnrm \leftarrow \max|r|$
9: **while** $Rnrm > Xnrm \times Cte$ **and** $Iter < IterMax$ **do**
10: **Solve:** $Ly = r$
11: **Solve:** $Uz = y$
12: $x \leftarrow x + r$
13: $r \leftarrow b - Ax$
14: $Xnrm \leftarrow \max|x|$
15: $Rnrm \leftarrow \max|r|$
16: $Iter \leftarrow Iter + 1$
17: **end while**

3.3 Hybrid RBT algorithm

In this section, we present an implementation of an RBT solver for hybrid CPU/accelerator architectures. We use two recursion levels for the randomization. The resulting computational cost of randomization step is $8n^2$ flops due to the block diagonal structure of the butterflies, as demonstrated in [30]. The RBT solver for hybrid architectures performs the following tasks:

1. Random generation and packed storage of the butterflies U and V on the CPU (host), while sending A to the device (accelerator) memory (if the size of the matrix A is not a multiple of 4, padding is added).

2. The packed U and V are sent from the host memory to the device memory.

3. The randomization is performed on the accelerator. The update of A is done in-place (no additional memory needed) on the device memory.

4. The randomized matrix is factorized with GENP, the panel factorization being performed on the CPU host and the update of the trailing submatrix on the accelerator.

5. We compute $U^T b$ on the device, then $A_r y = U^T b$ is also solved on the accelerator.

6. If necessary, iterative refinement is performed on the solution y using the accelerator.

7. We compute the solution $x = Vy$ on the device and the solution x is sent to the host memory.

Note that the step 4 is performed using routines of LU factorization with no pivoting adapted from the MAGMA routines of LU factorization with pivoting. These routines use the same method as described in Section 2.2.

The randomization step is performed as follows:

1. U and V are two N-by-N recursive butterfly matrices of depth two.

   We consider that A can be split into 4 blocks of same size as follows: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$

2. $U = U_2 \times U_1$ and $V = V_2 \times V_1$ with U_1, V_1 being two butterfly matrices and U_2, V_2 two matrices of the form $\begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix}$, where B_1 and B_2 are two $N/2$-by-$N/2$ butterfly matrices as illustrated in Equation 3.1.

3. We note $A_r^1 = U_2^T \times A \times V_2$.

   Since $A_r = U^T A V = U_1^T \times A_r^1 \times V_1$, we first apply U_2^T and V_2.

4. We compute $U_2^T \times A \times V_2 = \begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} \times \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_1' & 0 \\ 0 & B_2' \end{pmatrix} = \begin{pmatrix} B_1 A_{11} B_1' & B_1 A_{12} B_2' \\ B_2 A_{21} B_1' & B_2 A_{22} B_2' \end{pmatrix}$

   This step consists of four independent products of the form $U^T A V$ with depth-1 butterfly matrices of size $N/2$-by-$N/2$. We call the kernel used for this product `Elementary multiplication`.

5. We then apply U_1^T and V_1 to A_r^1 to obtain A_r, consisting in using `Elementary multiplication` on N-by-N matrices.

When using the RBT solver, the bulk of the computation is done by the GENP factorization (experiments show that the randomization represents less than 4% of the global computational time). Up to now, the no pivoting version of the LU factorization has been the most efficient. Thus, we expect that the Gflop/s performance of the RBT solver will provide us with an upper bound for other LU solvers on hybrid architectures.

3.4 RBT solver using Graphic Process Units

In this section we present details on the optimized implementation of the randomization part of the RBT solver using GPU. Then we give performance results of this implementation that has been developed for the MAGMA library.

As mentioned in Section 1.3.2, even if pivoting improves the numerical stability of the LU factorization, the data movement and search for the pivots are time consuming. In Figure 3.2, we show the overhead of the pivoting phase in an LU factorization (partial pivoting) using the MAGMA library with an NVIDIA Tesla K20 GPU as accelerator. We observe that pivoting (selection of pivots and swap of rows) takes more than 20% of the total computational time for matrices of size smaller than 10000. For large enough matrices, most of the computational time is spent in matrix-matrix products (DGEMM) on the GPU, reducing the overhead of the pivoting.

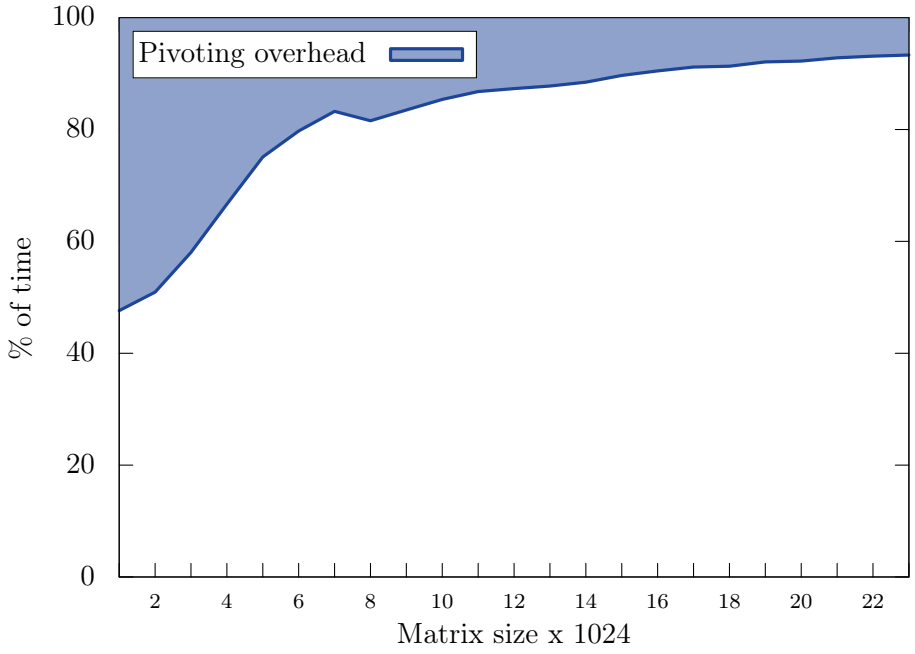


Figure 3.2: Pivoting cost of the LU factorization on GPU.

This result confirms that a solver based on a factorization with no pivoting would outperform the classically used LU factorization with partial pivoting.

Our RBT solver exists for all precisions used in LAPACK (simple, double, simple complex and double complex) and we integrated it in a recent release[1] of the MAGMA library. It includes GPU-based randomization routines, a no pivoting factorization routine, and iterative refinement on GPU.

[1] see http://icl.cs.utk.edu/magma/news/news.html?id=351

3.4.1 Implementation

On hybrid CPU/GPU architectures, the RBT solver is performed as described in Section 3.3. In Algorithm 6, we give details on the routines that perform the randomization of the matrix A.

Algorithm 6 RBT with two recursions

Input: A a pointer to the matrix A on the GPU.
Input: U a pointer to the matrix U stored as a vector on the GPU.
Input: V a pointer to the matrix V stored as a vector on the GPU.
Input: N the size of the matrix A
Result: $A \leftarrow U^T AV$

1: $block\_height \leftarrow 32$
2: $block\_width \leftarrow 4$
3: **Define** a grid of threads per block, size: $(block\_height, block\_width)$
4: **Define** a grid of blocks, size: $(\frac{N}{4 \times block\_height}, \frac{N}{4 \times block\_width})$
{Assuming N is divisible by $4 \times block\_height$ and $4 \times block\_width$}
{ All GPU kernels are called with the threads and grid dimensions defined before the call}
5: **Call:** Elementary Multiplication(A, $\&U(N)$, $\&V(N)$, $N/2$)
6: **Call:** Elementary Multiplication($\&A(0, N/2)$, $\&U(N)$, $\&V(N + N/2)$, $N/2$)
7: **Call:** Elementary Multiplication($\&A(N/2, 0)$, $\&U(N + N/2)$, $\&V(N)$, $N/2$)
8: **Call:** Elementary Multiplication($\&A(N/2, N/2)$, $\&U(N+N/2)$, $\&V(N+N/2)$, $N/2$)
9: **Redefine** a grid of blocks, size: $(\frac{N}{2 \times block\_height}, \frac{N}{2 \times block\_width})$
{Assuming N is divisible by $2 \times block\_height$ and $2 \times block\_width$}
10: **Call:** Elementary Multiplication(A, U, V, N) {Applying level 1 recursion}

This function applies the depth-two RBT to the matrix A by processing first each $N/2$-by-$N/2$ quarter block of the matrix and then applying the level one recursion to all the N-by-N matrix as described in Section 3.3. The application of the level two of RBT consists in calling the `Elementary Multiplication` kernel on each quarter part of the matrix. This is due to the block diagonal structure of the butterfly matrix. Each `Elementary Multiplication` kernel is performed with one GPU thread per element.

The `Elementary Multiplication` kernel performs $A \leftarrow U^T AV$, where U and V are size N vectors containing the coefficients of depth one random butterfly matrices as described in Section 3.2.

Algorithm 7 shows the details of the GPU implementation of the `Elementary Multiplication` kernel.

Algorithm 7 GPU Kernel: Elementary Multiplication(A, U, V, N)

for each thread block of size $bsize.x \times bsize.y$ of coordinates $b.x$ and $b.y$ **do**
  for each Thread of coordinates $t.x$ and $t.y$ in the block **do**
    $idx \leftarrow b.x \times bsize.x + t.x$
    $idy \leftarrow b.y \times bsize.y + t.y$
    if $idx < N/2$ **and** $idy < N/2$ **then**
      Declare 4 shared memory arrays: $U_1[bsize.x]$, $U_2[bsize.x]$, $V_1[bsize.y]$, $V_2[bsize.y]$
      $U_1(t.x) \leftarrow U(idx)$
      $U_2(t.x) \leftarrow U(idx + N/2)$
      $V_1(t.y) \leftarrow V(idy)$
      $V_2(t.y) \leftarrow V(idy + N/2)$
      Synchronize the threads in the block
      $a_{00} \leftarrow A(idx, idy)$
      $a_{01} \leftarrow A(idx, idy + N/2)$
      $a_{10} \leftarrow A(idx + N/2, idy)$
      $a_{11} \leftarrow A(idx + N/2, idy + N/2)$
      $b_1 \leftarrow a_{00} + a_{01}$
      $b_2 \leftarrow a_{10} + a_{11}$
      $b_3 \leftarrow a_{00} - a_{01}$
      $b_4 \leftarrow a_{10} - a_{11}$
      $A(idx, idy) \leftarrow U_1(t.x) \times V_1(t.y) \times (b_1 + b_2)$
      $A(idx, idy + N/2) \leftarrow U_1(t.x) \times V_2(t.y) \times (b_3 + b_4)$
      $A(idx + N/2, idy) \leftarrow U_2(t.x) \times V_1(t.y) \times (b_1 - b_2)$
      $A(idx + N/2, idy + N/2) \leftarrow U_2(t.x) \times V_2(t.y) \times (b_3 - b_4)$
    end if
  end for
end for

We use shared memory arrays for each block of threads to store the coefficients of U and V relative to this block and thereby improve the efficiency of the access to these elements.

3.4.2 Performance

The following section presents performance results for the RBT solver in MAGMA with GPU. The experiments were carried out on a system composed of:

- a GPU, NVIDIA Kepler K20, with 2496 CUDA cores running at 706 MHz and 4800 MB of memory
- a multicore host composed of two Intel Xeon X5680 processors, each with 6 physical cores running at 3.33 GHz, and a Level 3 memory cache of 12 MB.

The CPU parts of our code are performed using the multithreaded Intel MKL library.

In Figure 3.3, we can see that, the CUDA implementation of our RBT solver (either with or without iterative refinement) outperforms the classical LU factorization with partial pivoting from MAGMA. For large enough matrices (from size 6000) the obtained performance is about 20-30% faster than the solver based on Gaussian elimination with partial pivoting.

In our experiments, when we enable iterative refinement, only one iteration is performed which is generally enough to improve the computed solution giving an accuracy similar to the one obtained with partial pivoting. The iterative refinement is performed on the GPU and requires $\mathcal{O}(n^2)$ extra floating point operations, which in our case has no significant impact on the performance.

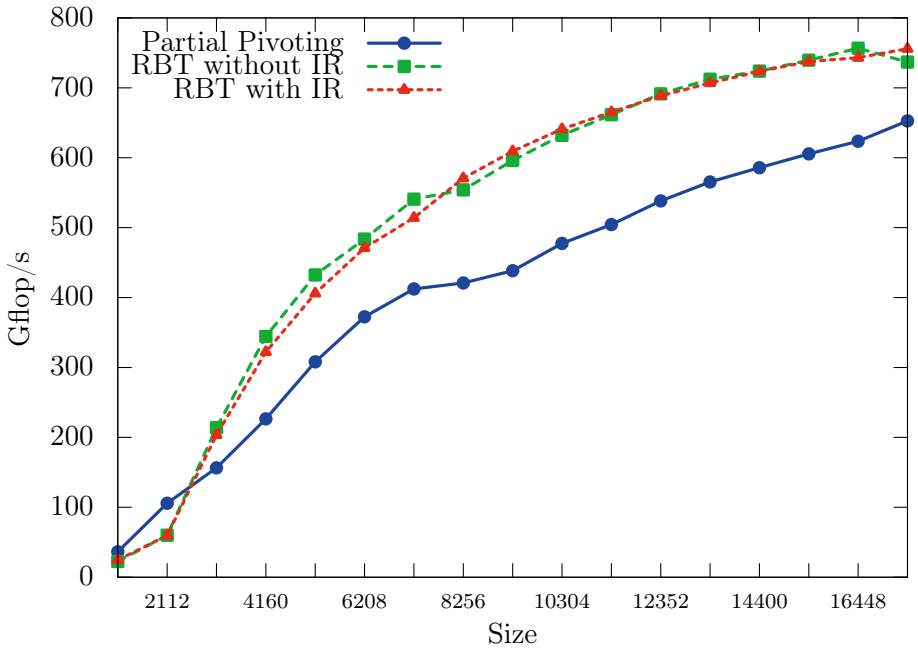


Figure 3.3: Performance of the RBT solver on GPU.

In Figure 3.4, we notice that the time required to perform the randomization is less than 4% for small matrices and becomes less than 2% for bigger matrices. This is due to the low computational cost of the randomization ($4n^2$ flops) and to our optimized implementation that use the capabilities of the GPU accelerator.

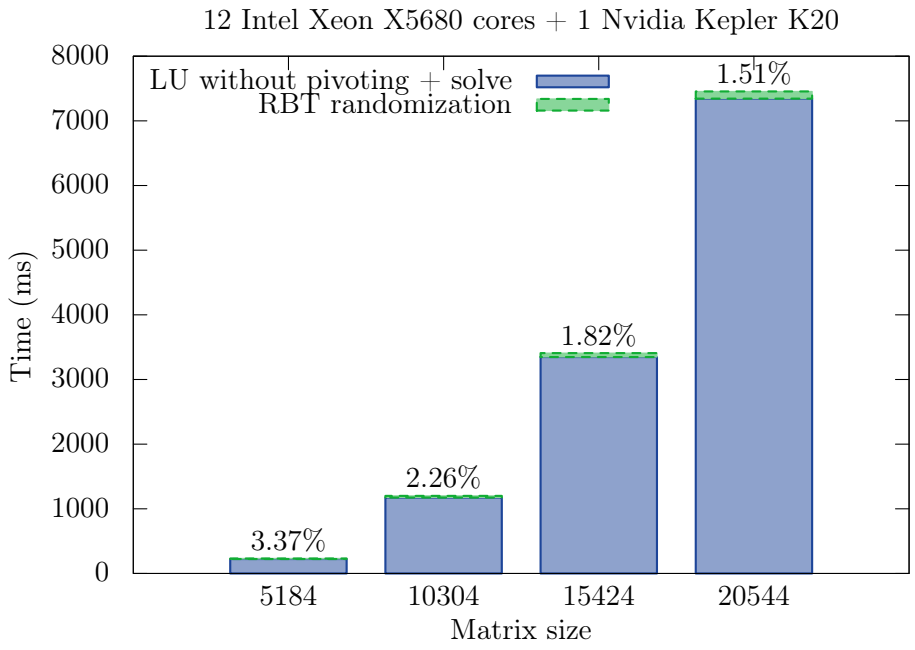


Figure 3.4: Time breakdown of the RBT solver on GPU.

3.5 RBT solver using Intel Xeon Phi coprocessors

Similarly to the previous section, we present our implementation of the RBT solver on Intel Xeon Phi coprocessor and some performance result. This solver and all the required functions are part of the MAGMA MIC library since version 1.3. This includes the following routines: no pivoting LU fatorization, no pivoting solver, randomization routines, iterative refinement and the global solver based on RBT + LU without pivoting + iterative refinement. These routines are all available in the four standard precisions (single, double, single complex, double complex).

In the following, we give more details on the randomization routine used in the RBT solver.

Figure 3.5, shows the time cost of the partial pivoting. The difference of performance compared to the GPU implementation (see Figure 3.2), is understandable. Experiments have shown that the Xeon Phi version of the factorization need a greater amount of data than the GPU to be efficient. Indeed, for a matrix size of order 6000, our solver gives performance around 200 Gflop/s for the Xeon Phi whereas it is around 500 Gflop/s with the GPU. When we increase the size of the problem, the performance of both versions tend towards 800 Gflop/s (for double precision). For small matrices, the pivoting overhead is proportionally smaller than on GPU but the global performance of the solvers for such matrices is worse regardless of the pivoting strategy used. The "spikes" that can be observed for some matrix sizes correspond to changes in the panel size used, creating a gap in the performance.

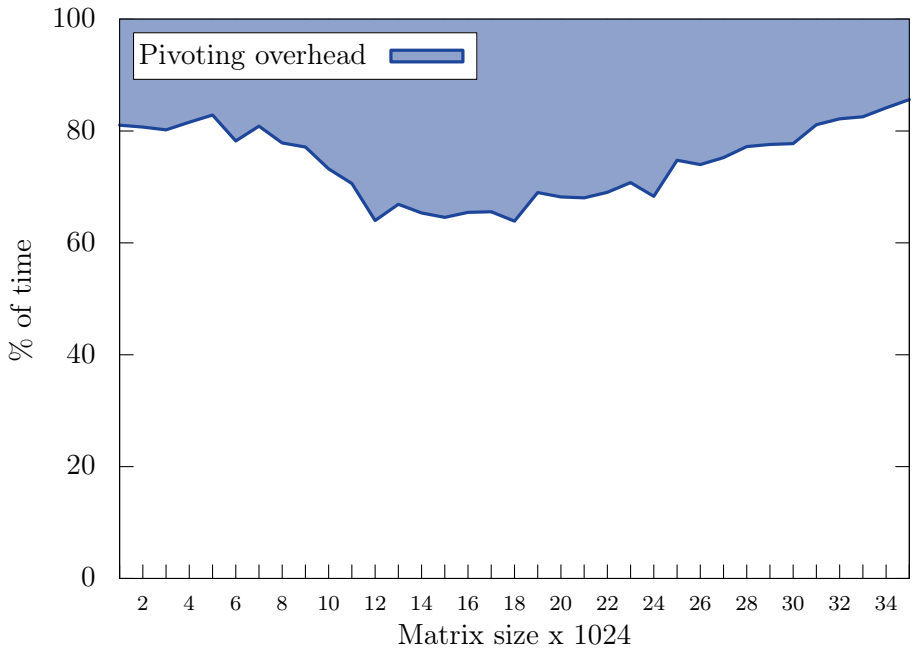


Figure 3.5: Pivoting cost in the LU factorization on Xeon Phi.

3.5.1 Implementation

Algorithm 8 presents the implementation of the RBT randomization, using depth two butterfly matrices. It is similar to its GPU counterpart, except that there are no blocks or threads to deal with inside this function.

Algorithm 8 RBT with two recursions

Input: A a pointer to the matrix A on the Phi.
Input: U a pointer to the matrix U stored as a vector on the Phi.
Input: V a pointer to the matrix V stored as a vector on the Phi.
Input: N the size of the matrix A
Result: $A \leftarrow U^T AV$

1: **Call:** Elementary Multiplication Phi(A, $\&U(N)$, $\&V(N)$, $N/2$)
2: **Call:** Elementary Multiplication Phi($\&A(0, N/2)$, $\&U(N)$, $\&V(N+N/2)$, $N/2$)
3: **Call:** Elementary Multiplication Phi($\&A(N/2, 0)$, $\&U(N+N/2)$, $\&V(N)$, $N/2$)
4: **Call:** Elementary Multiplication Phi($\&A(N/2, N/2)$, $\&U(N + N/2)$, $\&V(N + N/2)$, $N/2$)
5: **Call:** Elementary Multiplication Phi(A, U, V, N) {Applying level one recursion}

The `Elementary multiplication Phi` described in Algorithm 9 uses SIMD instructions to improve the performance of each core and OpenMP to handle thread parallelism between cores. This algorithm is well adapted to the SIMD program-

ming model as the dependencies between the data are separated by a large number of values. In Algorithm 9, we consider that we are working with double precision floating point numbers, each of them using 64 bits. This explains why 8 values are stored in each 512-bits SIMD vector. When using 32 bits reals, 16 values are stored per vector. For complex numbers, 8 numbers are store in single precision and 4 in double.

We take advantage of the SIMD capabilities of the Intel Xeon Phi coprocessor by using the low level Knight's Corner intrinsics set of instructions. The use of the intrinsics allows to use the assembly SIMD instructions with C style functions.

Algorithm 9 Phi Kernel: Elementary multiplication Phi(A, U, V, N)

OpenMP parallel for
for $i = 0$ **to** $N/2$ **do**
  Declare V_1 and V_2 two 512-bit vector registers.
  Set all values of V_1 with $V(i)$
  Set all values of V_2 with $V(i + N/2)$
  for $j = 0$ **to** $N/2$ **step** 8 **do**
    Declare a_{00}, a_{01}, a_{10} and a_{11} four 512-bit vector registers.
    LOAD 8 values from $A(i, j)$ in a_{00}
    LOAD 8 values from $A(i, j + N/2)$ in a_{01}
    LOAD 8 values from $A(i + N/2, j)$ in a_{10}
    LOAD 8 values from $A(i + N/2, j + N/2)$ in a_{11}
    Declare b_1, b_2, b_3 and b_4 four 512-bit vector registers.
    $b_1 \leftarrow$ **ADD**(a_{00}, a_{01})
    $b_2 \leftarrow$ **ADD**(a_{10}, a_{11})
    $b_3 \leftarrow$ **SUB**(a_{00}, a_{01})
    $b_4 \leftarrow$ **SUB**(a_{10}, a_{11})
    Declare U_1 and U_2 two 512-bit vector registers.
    LOAD 8 values from $U(j)$ in U_1
    LOAD 8 values from $U(j + N/2)$ in U_2
    $a_{00} \leftarrow$ **MUL**(U_1, **MUL**(V_1, **ADD**(b_1, b_2)))
    $a_{01} \leftarrow$ **MUL**(U_1, **MUL**(V_2, **ADD**(b_3, b_4)))
    $a_{10} \leftarrow$ **MUL**(U_2, **MUL**(V_1, **SUB**(b_1, b_2)))
    $a_{11} \leftarrow$ **MUL**(U_2, **MUL**(V_2, **SUB**(b_3, b_4)))
    STORE 8 values from a_{00} at $A(i, j)$
    STORE 8 values from a_{01} at $A(i, j + N/2)$
    STORE 8 values from a_{10} at $A(i + N/2, j)$
    STORE 8 values from a_{11} at $A(i + N/2, j + N/2)$
  end for
end for

3.5.2 Performance

We present the performance results of the RBT solver in MAGMA when used with the Intel Xeon Phi coprocessor as an accelerator. The experiments were carried out using the same multicore host as in Section 3.4.2 (two Intel Xeon X5680) but with an Intel Xeon Phi coprocessor 7120 with 61 cores running at 1.238 GHz, with 16 GB of memory. The cores have 30.5 MB of combined L2 cache memory. We mention that each core can manage 4 threads by hyper threading. For the experiments, a total of 240 threads is used.

For these experiments, we were able to perform tests on bigger matrices compared to the GPU version. This is due to the larger size of the Intel Xeon Phi memory.

In Figure 3.6, we notice that the Intel Xeon Phi version performs up to 65% faster than the solver using partial pivoting without iterative refinement, and only 26% faster with iterative refinement (which is not yet optimized for Intel Xeon Phi).

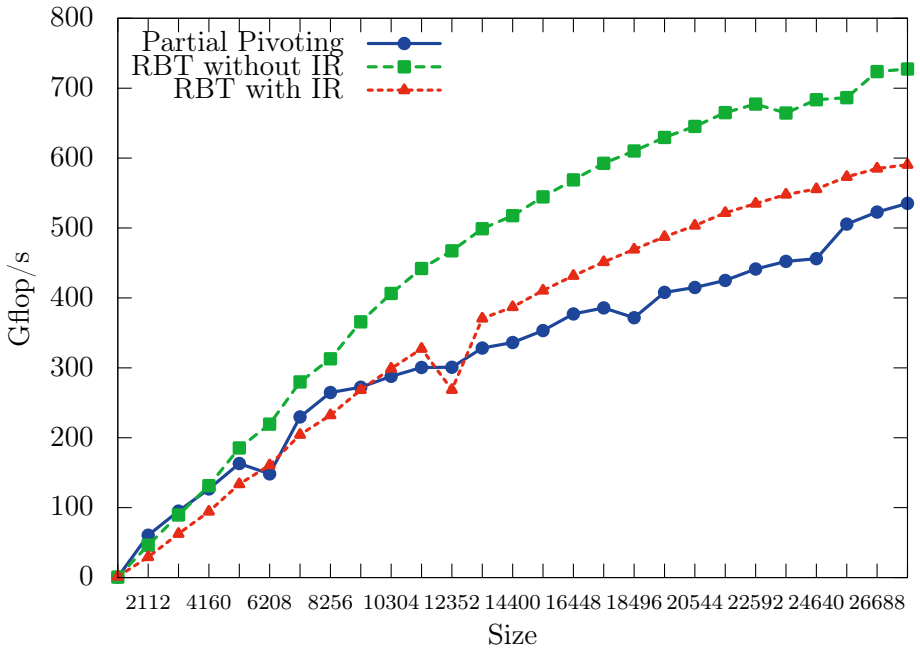


Figure 3.6: Performance of the RBT solver on Xeon Phi coprocessor.

In Figure 3.7, we observe that the randomization requires less than 3% of the total time and even less than 1% for big matrices. We recall that the randomization performed on the Intel Xeon Phi has been optimized using MIC SIMD instructions and OpenMP.

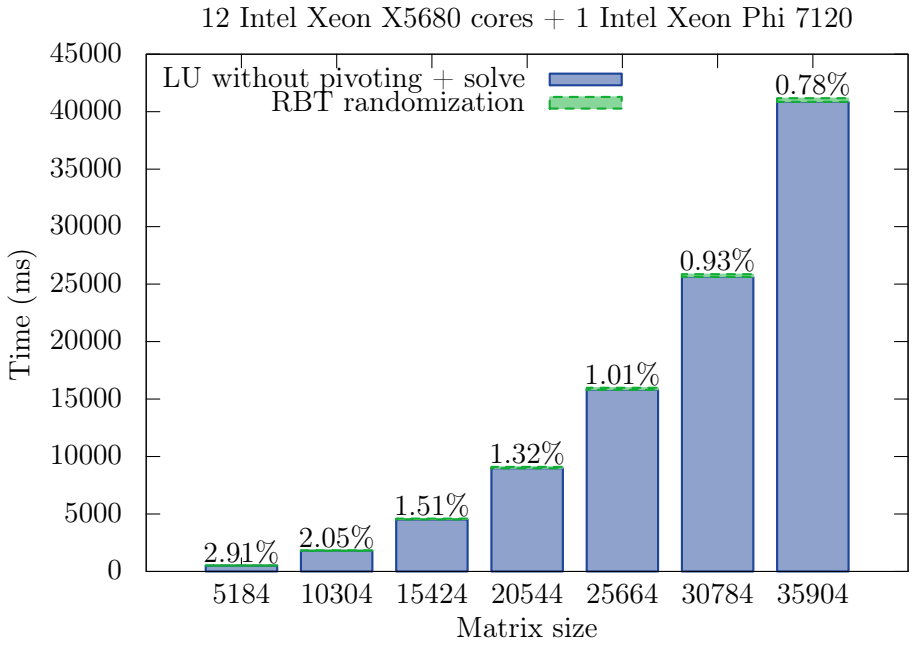


Figure 3.7: Time breakdown of the RBT solver on Xeon Phi coprocessor.

3.6 Conclusion of Chapter 3

In this chapter, we have presented two implementations of the RBT solver using accelerators. One is based on GPU and the other an Intel Xeon Phi. Both methods have been implemented for the MAGMA library and we have shown that they significantly outperform the reference solver based on the LU factorization with partial pivoting. Due to the optimized implementation of the randomization, the overhead for randomizing the system is negligible compared to the computational cost of the whole solver.

Ongoing work includes optimizing the iterative refinement on Intel Xeon Phi. Also, we plan to adapt RBT to hybrid architectures to solve symmetric indefinite systems, and to solve multiple small systems at the same time using batched solvers [99].

In the next chapter, we present thread and data placement methods in order to improve the performance of dense linear solvers when implemented on NUMA architectures.

Chapter 4

Locality optimization for NUMA architectures

Contents

4.1 Using NUMA architectures for dense linear systems

On modern parallel systems, the main memory bandwidth plays a crucial role in high-performance computing (HPC) applications. On shared memory parallel computers, a large number of processors work on a common, shared physical address space. There are two types of shared memory systems that propose similar functionalities to the programmer but have different performance in terms of main memory access.

Unified Memory Access (UMA) systems consist of a single memory bank for which the latency and bandwidth are the same for all threads, regardless of the memory location. The downside of UMA systems is that, when many application threads are trying to access the main memory simultaneously, bandwidth bottlenecks can occur. To overcome this problem of scalability, architectures referred to as ccNUMA (cache coherent Non Uniform Memory Access) are commonly used in clusters of nodes. Recently, the ccNUMA has been adapted inside multicore nodes (see, e.g., [100] and the references therein). On ccNUMA systems, the memory is physically distributed but logically shared. The mechanism is transparent from the programmer point of view, the required protocols being handled by the hardware (*e.g.* HyperTransport for AMD and QuickPath for Intel). Each bank of memory is associated with a set of cores and this association forms a NUMA node. Due to this physical distribution, the performance of memory accesses varies depending on the

mutual location of a given thread and the memory bank that the thread accesses. Accessing the remote memory banks may become slow and, if a lot of threads are used, this will affect the application scalability [101]. When using HPC applications on ccNUMA systems, we face two main difficulties. The first one is the locality problem. It happens when a thread located on a node accesses data stored in the memory bank of another node. This kind of nonlocal transfer can hurt performance. The second problem is contention, which occurs when two threads located on different nodes access memory in another node, and thus, fight for memory bandwidth. For each thread, the access to data should be restricted to its own node to avoid these two problems. If no particular data placement is proposed, the default memory affinity policy of the operating system is used. In most Linux-type operating systems, the default policy (called *first touch*) places the data in the memory node that is local to the thread that is writing the data first. This ensures fast access for the thread inside the node regardless of the other threads accessing the data [102]. In a multithreaded application, the fact that the master thread usually initializes multiple shared data structure can exacerbate the problem (all these shared data structures will be allocated in the same node as the master thread). This problem can be approached by using software tools, such as the `libnuma` library [103] or the `likwid` software [104], that provide user interfaces to allocate memory into the desired nodes [103] or by initializing the data by multiple (possibly all) threads. The Servet Benchmark Suite [105] also provides an API to handle threads mappings based on communication or memory performance. Even if data locality is respected, the thread scheduling is important. If the scheduler ignores the locality information, the effect of caches is reduced. Switches into uncached process contexts will cause cache and TLB misses and cache line invalidations for the other processes [106]. The cost of thread scheduling can be reduced by moving thread management and synchronization to the user level [107].

In this chapter, we study the effect of NUMA on the solution of dense general linear systems. To solve square linear systems $Ax = b$, the method commonly used is Gaussian Elimination with partial pivoting (GEPP) (see 1.3.3). Libraries, such as LAPACK [68], provide a block algorithm version of GEPP where the factorization is performed by iterating over blocks of columns (panels) (see 1.3.5). LAPACK has been redesigned to use heterogeneous systems of multi/manycore CPUs and accelerators, such as GPUs. Examples of the redesign are PLASMA [79] and MAGMA [85], which take advantage of current multicore and hybrid multicore/GPU architectures [86]. In a classical LU factorization, the panel is first factored and then the trailing submatrix is updated using level 3 BLAS routines for high performance [108]. The update consisting of matrix-matrix products performed by the GPU is very efficient, making the panel factorization the bottleneck of the performance [82]. Indeed, due to its data access pattern the panel factorization is widely affected by memory access performance. By reducing the time of the panel factorization, we can improve the performance of the overall computation.

In the following we show how a proper placement of the threads and memory on a NUMA architecture can improve the performance of the panel factorization

and consequently accelerate the global LU factorization as it is implemented in the MAGMA library for multicore/GPU systems. The chapter is organized as follows. In Section 4.2, we describe the application context in which this work takes place. Then in Section 4.3, we describe different strategies for thread pinning and data placement on NUMA architectures. Section 4.4 presents the experimental setup and performance results of LU factorization on a given platform using the different placement strategies given in Section 4.3. Concluding remarks are given in Section 4.5.

4.2 Application context

We consider here the hybrid LU factorization (right looking factorization [1, p. 85]) as implemented in the MAGMA library and detailed in Section 2.2.

In the remainder, we will consider two MAGMA implementations for the LU factorization. These two versions differ mainly in the way the panel is factored. In the first version, the panel is factored using GEPP (partial pivoting) while the second version does not pivot since it also uses randomization as a preprocessing to avoid pivoting [30]. We point out that in both MAGMA implementations the panel is factorized as a BLAS 3 algorithm where we consider an inner panel (factored using BLAS 2) inside the global panel. The size of this inner panel is set to 128 for the no pivoting version, and cannot be tuned for the partial pivoting when we use an MKL [39] implementation. Note that larger size of the panel results in more BLAS 3 operations, and thus, increasing the computation-to-memory access ratio in the panel factorization.

Due to the search for the pivot and to the subsequent row interchange, GEPP performs a lot of memory accesses, whereas they are minimal for the version without pivoting. In the following we focus on the panel factorization, because its memory-bound characteristics make it particularly dependent on NUMA.

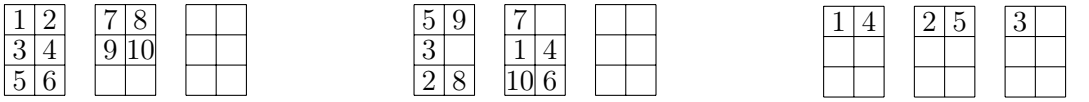
4.3 Placement strategies

In this section we describe how threads may be bound to cores and how data may be placed in memory, which may be achieved using the following tools.

- Before each execution the data are placed in the node using the `mbind()` function from the `libnuma` [103] library.
- The threads may be pinned to the cores using the `sched_setaffinity()` Unix function, the `likwid` [104] or `numactl` [103] tools.
- Before each execution, using the same tools, the data may be placed in the nodes to which the threads are bound.

For the thread pinning, we consider the following strategies, which are illustrated in Figure 4.1 by considering three nodes of six cores. For all the different strategies

the data are interleaved over the memory banks of the NUMA nodes used (i.e., the data are spread in a round-robin fashion in the memory pages across all the nodes).



(a) Sequential pinning 1 (b) Sequential pinning 2 (c) Cyclic pinning

Figure 4.1: Examples of pinning methods

No pinning: The threads and data are assigned automatically by the system, without manual intervention. We refer to this strategy as `noPin`.

Sequential pinning 1: The number of threads assigned to a node corresponds to its number of cores. When a given node is full while more threads need to be placed, the next thread will be "spilled" to the next node. In a sense, nodes are provided one by one for the thread placement purposes. An example is given in Figure 4.1a, where we use 10 threads. Note that the thread placement *within* a node is not fixed explicitly and may be governed by the application. For example, the MKL Intel mathematical library may assign threads to cores dynamically. We refer to this strategy as `seqPin1`.

Sequential pinning 2: To avoid the problem of load balancing among the nodes when the number of threads is not a multiple of the number of cores per node, we allow the application to place threads evenly on several nodes at once. Specifically, these "occupied" nodes are taken as the first nodes that can accommodate all the threads. In Figure 4.1b, the application will place them on the first two nodes, such that each node may have a free core. We refer to this strategy as `seqPin2`.

Cyclic pinning: The threads are cyclically placed onto all the nodes allotted to the application in a round-robin manner. For example in Figure 4.1c, we use five threads in three nodes. Note that, although this approach is a rather load-balanced (the number of threads in each node may differ by one at most), it is not compact in terms of processing power. On the other hand, as will be shown later, its memory availability may be attractive. We refer to this strategy as `cycPin`.

Since the data are interleaved only among the nodes that are effectively used, sequential pinnings result in mostly local memory accesses. The memory accesses may incur extra latency for the cyclic pinning. However, for the cyclic pinning, there will be less competition among threads inside a node to access L3 cache. Moreover, the global amount of L3 cache available for all the threads will be larger for `cycPin`

because this strategy uses more nodes. Note that if the number of threads used is the same as the number of cores available, the sequential and the cyclic pinning are equivalent in terms of data locality. In [109] and [110], we have investigated more sophisticated strategies for memory binding to NUMA nodes, which improved performance on sparse matrix computations with irregular matrix and vector access patterns. For the dense matrices and dynamic thread placement within a node, interleaving data in memory is sufficient mainly due to the dynamic nature of the thread placement by MKL.

4.4 Application to LU factorization

4.4.1 Experimental framework

Our experiments have been carried out using a MagnyCours-48 system. This machine is composed of four AMD Opteron 6172 processors running at 2.1GHz with twelve cores each (48 cores total) and 128GB of memory. Each processor contains two NUMA nodes with 6MB of L3 cache per node. Thus, we have 8 NUMA nodes of 6 CPU cores and 16 GB of main memory each. The GPU device is an NVIDIA Fermi Tesla S2050 with 448 CUDA cores running at 1.15 GHz and 2687 MB memory.

For a thread accessing memory in the same NUMA node the relative distance to the memory (latency) is taken as 10 in the ACPI specification [111]. The relative distances between the nodes are reproduced in Table 4.1 as obtained by the `numactl` tool [103]. For example, if node 0 accesses data stored in node 3, the cost of this access is 2.2 times larger than if the data were stored in node 0. The 8 nodes are linked by HyperTransport links. When a thread pinned on a core accesses data, if the data is located inside the same NUMA node as the core, the relative cost to access the memory will be 10. If the data is located in the memory from a node directly connected by an HyperTransport link the cost will be 16. If the memory is in a node that is not directly linked to the current then the cost will be 22 (the data have to pass through 2 links).

Table 4.1: Node distances with respect to NUMA accesses.

node	0	1	2	3	4	5	6	7
0:	10	16	16	22	16	22	16	22
1:	16	10	22	16	22	16	22	16
2:	16	22	10	16	16	22	16	22
3:	22	16	16	10	22	16	22	16
4:	16	22	16	22	10	16	16	22
5:	22	16	22	16	16	10	22	16
6:	16	22	16	22	16	22	10	16
7:	22	16	22	16	22	16	16	10

Figure 4.2 gives a representation of the architecture used in our experiments. The thick arrows represent the memory access inside a node (relative distance normalized

as 10) and the thin arrows (*e.g.* between node 0 and node 1) represent memory access to the node connected by this arrow (relative distance 16). When there is no arrow between two nodes (*e.g.* between node 0 and node 3), the cost to access the memory in this node will be 22.

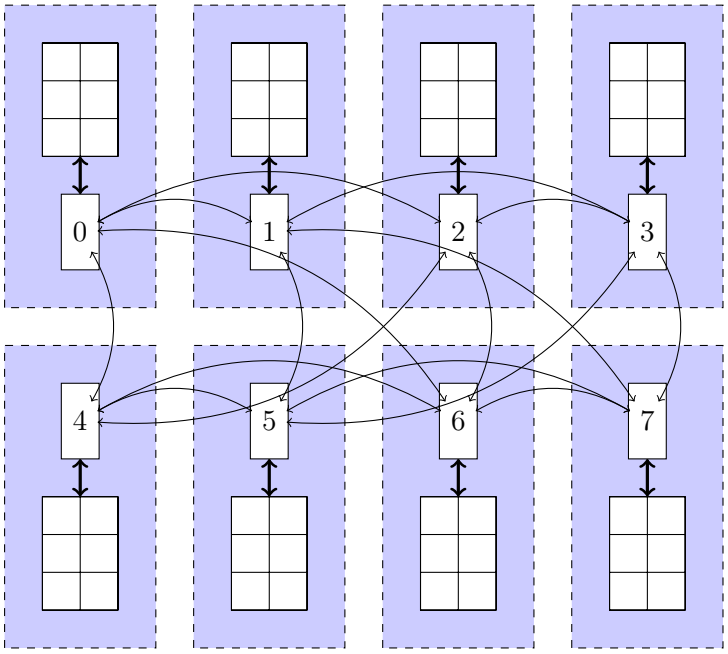


Figure 4.2: Architecture representation (8 nodes, 6 cores each)

We suppose now that the threads are scattered on all the nodes and that data are interleaved on the nodes. Then, assuming that each thread performs the same number of memory accesses on each node, we can compute the average memory access cost as $(10 + 16 + 16 + 22 + 16 + 22 + 16 + 22)/8 = 17.5$ in each of the eight nodes since all the rows in Table 4.1 have the same entries but in a different order. Therefore, the average memory access cost is 1.75 times larger than in the case of only local accesses.

4.4.2 Performance for the panel factorization

We test the performance of an LU panel factorization. This performance is expressed in Gflops/s. We measure it by summing the total number of flops executed in factoring successively each panel throughout the factorization and dividing it by the time spent in all the panel factorizations. The algorithms that we consider in our experiments are LU with partial pivoting and LU with no pivoting. The former uses the LAPACK implementation of GEPP (routine `dgetrf`) while the latter is a panel factorization with no pivoting (used in e.g., [93]), both linked with the multi-threaded BLAS from MKL.

In Figures 4.3 to 4.5, we compare the performance resulting from the different strategies of thread placements. The sequential pinning shown in the legend

corresponds to the `seqPin2` strategy as described in Section 4.3, which provides a better load balance than `seqPin1`. For each type of placement, we measure the performance using a number of threads varying from 1 to 48.

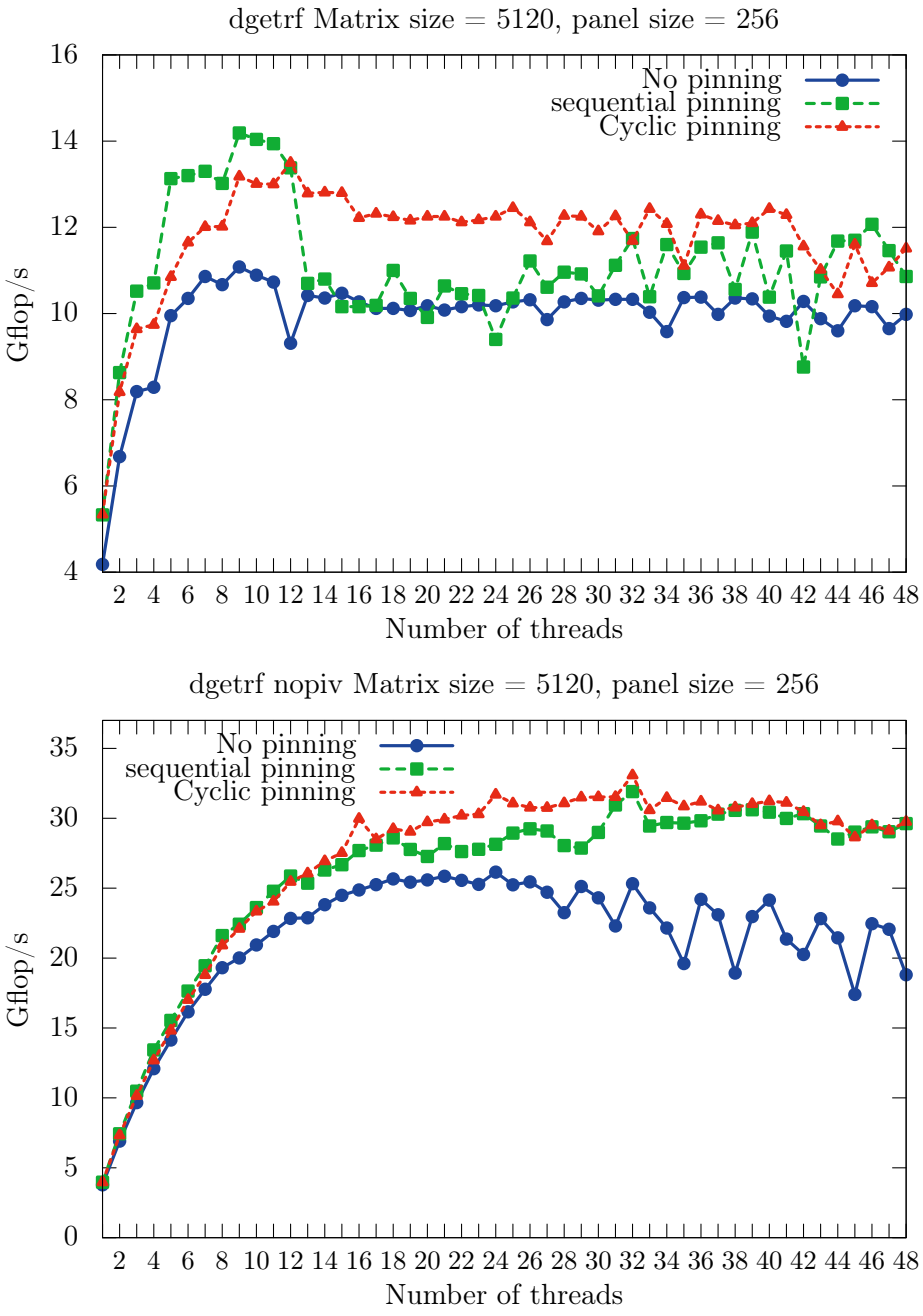


Figure 4.3: Performance of thread pinning strategies for LU panel factorization with pivoting (top) and no pivoting (bottom). Panel sizes: 5120×256.

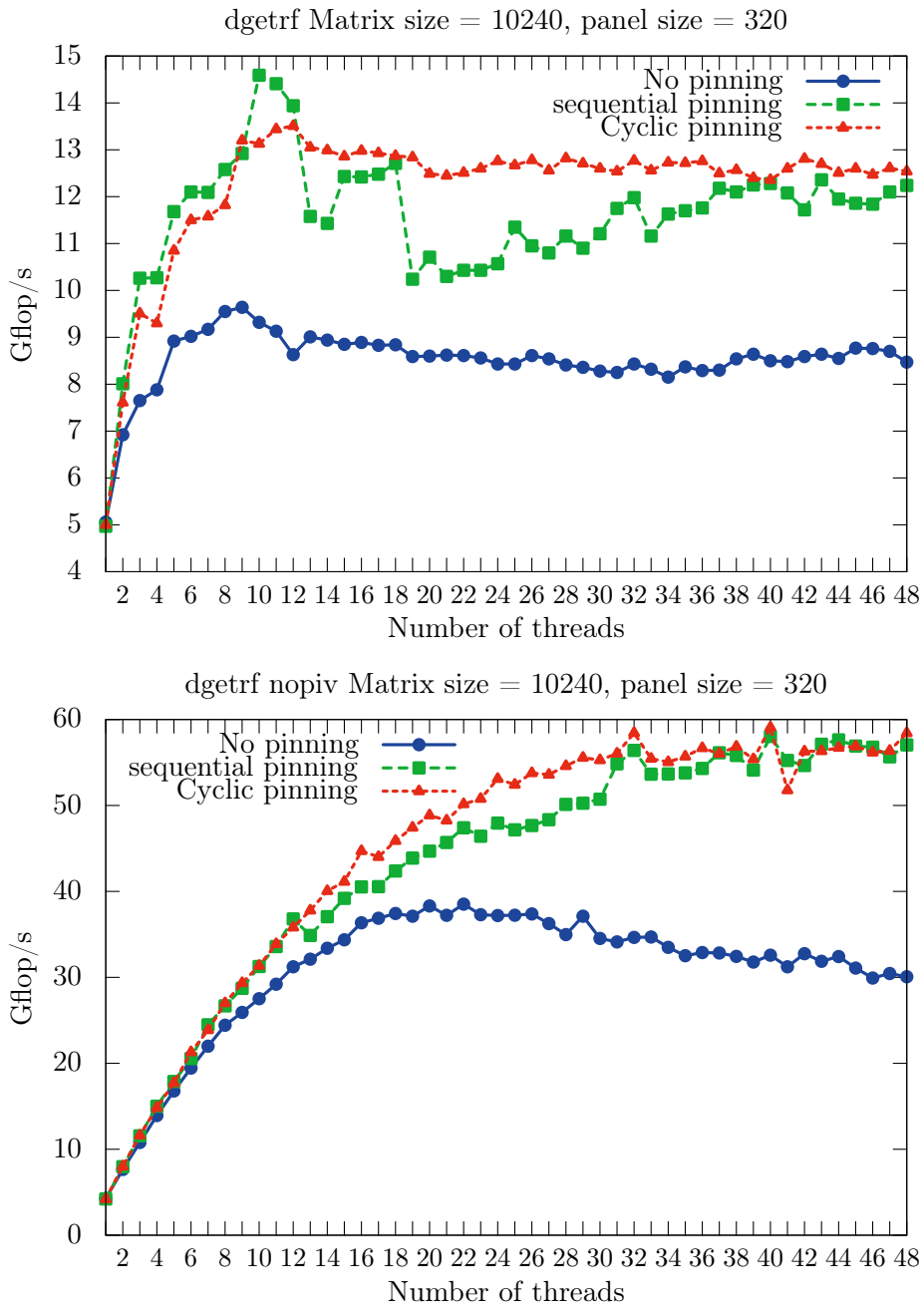


Figure 4.4: Performance of thread pinning strategies for LU panel factorization with pivoting (top) and no pivoting (bottom). Panel sizes: 10240×320.

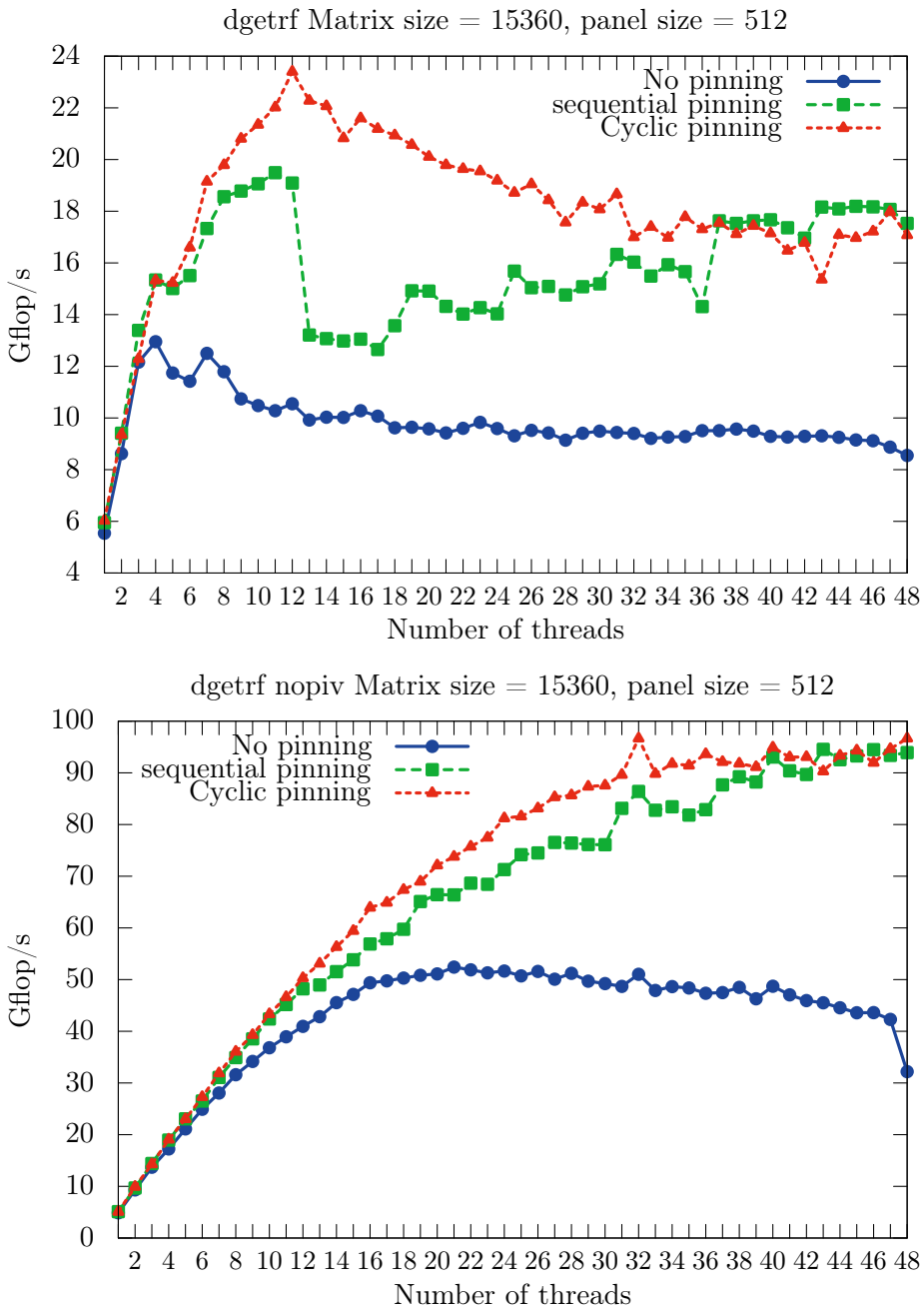


Figure 4.5: Performance of thread pinning strategies for LU panel factorization with pivoting (top) and no pivoting (bottom). Panel sizes: 15360×512.

When comparing the different types of pinning in Figures 4.3 to 4.5, we are interested in the peak performance obtained by each strategy and by the number of threads that enables us to obtain this rate. Indeed, since the scalability of the panel factorization is limited to a certain number of threads, it is not always recommended

to use all the CPU cores available. In particular, using the 48 cores available in our experiments is never the most efficient solution. A first comment is that, as expected, the pivoting LU algorithms outperforms the nonpivoting one with at least a factor 4. This is consistent with the results obtained and analyzed in [93, 25] and confirms that the communication overhead due to pivoting is critical for factorizing the panel [30, 26]. We observe that the sequential and cyclic pinnings give better results than `noPin`, and they are similar for larger numbers of threads (in the range 40-48).

For the partial pivoting case (`dgetrf`), the sequential pinning applied to a problem of size 10240 gives better performance than the cyclic pinnings for small thread counts due to a better data locality (i.e., because fewer NUMA nodes are involved). In our experiments, the best performance for size 10240 is obtained using 10 threads and consequently two nodes. We use nodes 0 and 1 which gives, using Table 4.1, an average memory access cost of 13 $((10 + 16)/2)$. The cyclic pinning gives better performance for the problem of size 15360, since there are more BLAS 3 operations that take better advantage of the cache and require fewer main memory accesses, possibly in remote NUMA nodes. We observe that for the sequential pinning, we have a performance drop for some number of threads, due to the addition of a new NUMA domain, namely when the number of threads is a multiple of 6, reducing then the data locality.

For the no pivoting case (`dgetrf_nopiv`), `cycPin` provides the best performance for all problem sizes. As expected, `dgetrf_nopiv` is less affected by data locality than `dgetrf` since there is no search for pivots, and thus, fewer memory access. Thereby, cache is used more efficiently, which is favored by the `cycPin` strategy that may make more cache available to threads due to the use of more nodes than for `seqPin2` in general. For example, if only one node is used, the amount of L3 cache available for the threads will be, on our architecture, 6MB and all the threads on the node will have to share it. If all of the 8 nodes are used then the memory accesses will be more expensive but the cache memory available will be $8 \times 6 = 48$ MB. Moreover, on this system the latency of the L3 cache is 20 ns, whereas the latency of the memory (inside a same node) is 60 ns. We also mention that these behaviors (ratio of cache misses, number of memory accesses) have been confirmed by measurements using the PAPI [112] library.

4.4.3 Performance for the hybrid code

Let us evaluate the impact of the thread/data placement on a hybrid CPU/GPU LU factorization. In this case, as explained in Section 4.2, the panel is factored by the CPU while the updates are performed by the GPU. In these experiments, the CPU uses a fixed number of threads and the matrix size varies.

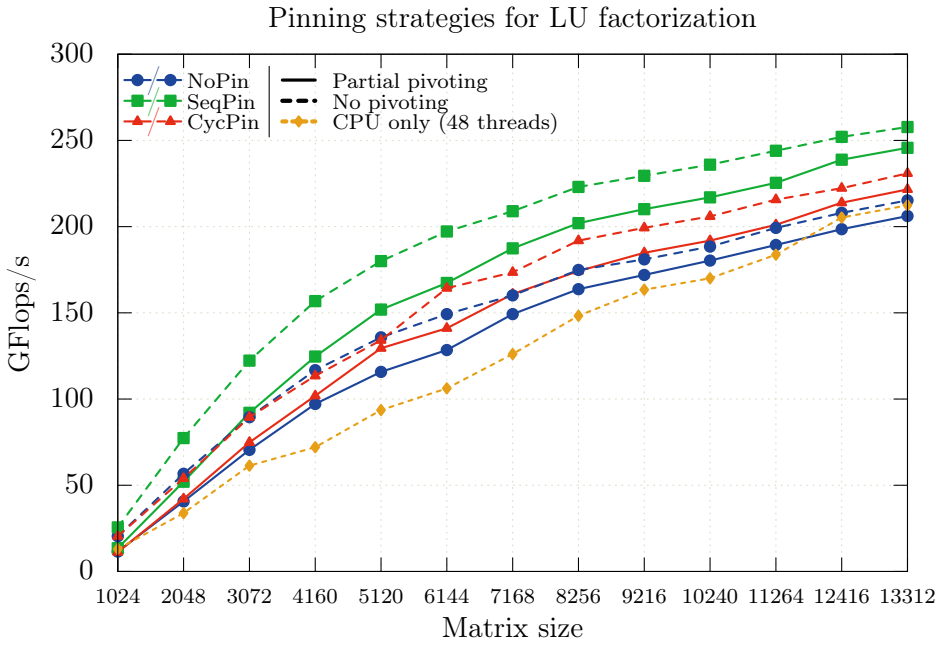


Figure 4.6: Performance for hybrid LU factorization with partial pivoting and no pivoting (12 threads).

Figure 4.6 compares the performance of the `seqPin2` and `cycPin` strategies with the `noPin` for the partial pivoting and no pivoting factorizations. The GPU was used along with the 12 CPU threads. For sake of comparison, we included in Figure 4.6 the performance for a CPU-only LU factorization using 48 threads (MKL implementation), the data being interleaved on all the nodes using the numactl tool. The NoPin curves represent the performance of the MAGMA codes without any modification. The SeqPin curves represent the performance of the MAGMA codes modified to have the threads pinned via the `seqPin2` strategy and the data placed only on the nodes that are actually used. The CycPin curves represent the performance with the threads pinned with the `cycPin` strategy with the data interleaved on the nodes.

We observe that the pinning methods outperform the `noPin` version and that for partial and no pivoting, the `seqPin2` strategy gives the best performance. Note that the difference of performance between the pivoting and no pivoting routines is smaller that for the panel factorization as depicted in Figure 4.3. Indeed, the update phase represents most of the computation and is performed by the GPU and the cost of the panel factorization has less impact on the global performance [93, 113]. Note also that asymptotically, when the matrix size increases and as mentioned in [30], the performance of the pivoting and no pivoting LU should be close because communication involved in pivoting becomes negligible compared to the $\mathcal{O}(n^3)$ computations for large dimensions.

4.5 Conclusion of Chapter 4

In this chapter, we studied different methods (referred to as sequential and cyclic pinning) to place threads and data for an LU factorization algorithm executed on a NUMA architecture using GPU accelerator. The two methods of placement improve the performance up to a factor 2 when compared with the default memory and thread placement. The choice of the most efficient method depends on the data pattern access of the algorithm and on the ability of the implementation to take advantage of cache memory. This choice is also influenced by the size of the problem. Subsequently to this work, we will develop a heuristic to choose automatically the best placement strategy. This technique has been implemented as an external function that can be easily applied to other algorithms with panel blocking.

A future work would be to implement this method in existing schedulers like QUARK [80] for the PLASMA [79] library, in order to allow an optimal placement of the data tiles on the different NUMA nodes. Part of the work described in this chapter has been published in [114].

Conclusion and future work

In the domain of high performance computing, parallel architectures provide more and more computational power, but are also more complex to program efficiently. Specifically, accelerators have become a predominant solution for current and future supercomputers due to the high computational power provided for a low energy cost. In this PhD thesis, we focused on designing efficient algorithms and software to solve dense linear systems on hybrid architectures. To reach these goals, we have proposed possible solutions to improve the performance of hybrid dense linear solvers using the LU factorization by choosing different strategies of pivoting, or using a randomization approach, and improving data and thread locality for NUMA architectures.

We proposed a comparison of the performance and numerical behavior of different algorithms for the LU factorization on hybrid architectures using GPUs as accelerators. We implemented a hybrid LU factorization using a communication-avoiding strategy of pivoting (H-CALU) with one GPU, and optimized the factorization of the panel. We also proposed a hybrid LU factorization with no pivoting using multiple GPUs, that could be used as part of a randomized solver and can be considered as an upper bound for performance. The H-CALU factorization outperforms the LU factorization using the partial pivoting included in MAGMA.

We also studied solvers based on Random Butterfly Transformations (RBT) as a preconditioning technique for solving dense linear systems, allowing to use an LU factorization with no pivoting, while remaining numerically stable in practice. We provide an RBT based solver for architectures using a GPU as accelerator and one using an Intel Xeon Phi coprocessor. These solvers are integrated in the MAGMA library for CUDA and Intel Xeon Phi. The optimized implementations of the accelerator kernels allow us to minimize the computational cost generated of the randomization and the resulting solvers significantly outperform the LU based solvers on partial pivoting.

Finally we proposed different methods for data and threads placement when performing an LU factorization on hybrid NUMA architectures using GPU. Due to the "memory bound" nature of the panel factorization on the CPUs, memory accesses on multiple NUMA nodes can greatly impact the performance. We then proposed two methods, the first one, called sequential pinning, which consists in pinning the threads on the minimum number of NUMA nodes possible while interleaving the data only between these nodes. This minimize the number of memory accesses in other NUMA nodes. The second one, called cyclic pinning, which consists in pinning the threads in the maximum number of nodes, interleaving the data between all the nodes used. With this method, we minimize the number of threads per multicore

processor, thus maximizing the availability of the shared last level cache memory for each thread. When used on the LU factorization, both of these methods outperform the standard first touch policy of data placement. The choice of the method depends on different factors which includes: the algorithm, the data pattern access and the size of the problem. These methods can be generalized and applied to other algorithms.

Perspectives

Future work includes implementing the H-CALU factorization for multiple GPUs in order to have scalable panel factorization and then to obtain good performance for large systems.

For the solvers based on the RBT randomization, ongoing works include applying RBT for hybrid architectures with accelerators to dense symmetric indefinite systems. Also we are developing an RBT solver for batched general dense linear systems on GPU. This can enable us to solve multiple small linear systems simultaneously on GPU. Our RBT solver for Intel Xeon Phi coprocessor also still needs an optimized version of the iterative refinement routine.

Concerning our work on data and thread placement for NUMA architectures, a future work could include the development of a heuristic to determine what would be the best pinning solution for a given task. Also we would like to implement our pinning solutions in schedulers such as QUARK for the PLASMA library to improve the NUMA awareness of the multicore solvers.

Bibliography

[1] J. Dongarra, I. S. Duff, D. Sorensen, and H. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, 1998. (Cited on pages vi, 11, 33 and 65.)

[2] L. Hodgkin. *A History of Mathematics: From Mesopotamia to Modernity*. Oxford University Press, 2005. (Cited on page 5.)

[3] A. Tucker. The growing importance of linear algebra in undergraduate mathematics. *College Mathematics Journal*, 24(1):3–9, 1993. (Cited on page 5.)

[4] I. B. Cohen. *Howard Aiken: Portrait of a computer pioneer*. MIT Press, 2000. (Cited on page 5.)

[5] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, jun 1945. Report prepared for U.S. Army Ordinance Department under Contract W-670-ORD-4926. (Cited on pages 5 and 18.)

[6] G. E. Moore et al. Cramming more components onto integrated circuits, 1965. (Cited on page 5.)

[7] A. Edelman. Large dense numerical linear algebra in 1993: The parallel computing influence. *International Journal of High Performance Computing Applications*, 7(2):113–128, 1993. (Cited on page 7.)

[8] J. K. Prentice. A Quantum Mechanical Theory for the Scattering of Low-Energy Atoms from Incommensurate Crystal Surface Layers. Technical report, New Mexico Univ., Albuquerque, NM (United States), 1992. (Cited on page 7.)

[9] A. Edelman. The first annual large dense linear system survey. *ACM SIGNUM Newsletter*, 26(4):6–12, 1991. (Cited on page 7.)

[10] X. Li and J. Demmel. SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003. (Cited on page 7.)

[11] P. Amestoy, J. Y. L'Excellent, F. H. Rouet, and M. Sid-Lakhdar. Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver. In *High-Performance Computing for Computational Science, VECPAR 2014, Eugene, Oregon, USA, 30/06/2014-03/07/2014*, http://www.laas.fr, 2014. LAAS. (Cited on page 7.)

[12] C. Fu, X. Jiao, and T. Yang. Efficient sparse LU factorization with partial pivoting on distributed memory architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 9(2):109–125, Feb 1998. (Cited on page 7.)

[13] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996. Third edition. (Cited on pages 7 and 44.)

[14] D. R. Kincaid and E. W. Cheney. *Numerical analysis: mathematics of scientific computing*, volume 2. American Mathematical Soc., 2002. (Cited on page 9.)

[15] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Clarendon Press Oxford, 1986. (Cited on page 9.)

[16] J. Malard. Threshold pivoting for dense LU factorization on distributed memory multiprocessors. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 600–607, New York, NY, USA, 1991. ACM. (Cited on page 9.)

[17] J. H. Wilkinson. Error Analysis of Direct Methods of Matrix Inversion. *J. ACM*, 8(3):281–330, July 1961. (Cited on page 9.)

[18] L. V. Foster. The growth factor and efficiency of Gaussian elimination with rook pivoting. *Journal of Computational and Applied Mathematics*, 86(1):177–194, 1997. (Cited on page 9.)

[19] L. V. Foster. Gaussian elimination with partial pivoting can fail in practice. *SIAM Journal on Matrix Analysis and Applications*, 15(4):1354–1362, 1994. (Cited on page 9.)

[20] J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, 1984. (Cited on page 10.)

[21] D.C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *Computers, IEEE Transactions on*, C-34(3):274–278, March 1985. (Cited on page 13.)

[22] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. Van De Geijn, and F. Van Zee. Design of scalable dense linear algebra libraries for multithreaded architectures: the LU factorization. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008. (Cited on page 13.)

[23] J. Demmel, L. Grigori, M.F. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report UCB/EECS-2008-89, EECS Department, University of California, Berkeley, Aug 2008. Current version available in the ArXiv at http://arxiv.org/pdf/0809.0101 Replaces EECS-2008-89 and EECS-2008-74. (Cited on page 14.)

[24] L. Grigori, J. Demmel, and H. Xiang. CALU: a communication optimal LU factorization algorithm. *SIAM J. Matrix Anal. and Appl.*, 32:1317–1350, 2011. (Cited on pages 14, 32 and 45.)

[25] S. Donfack, L. Grigori, and A. K. Gupta. Adapting communication-avoiding LU and QR factorizations to multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010. (Cited on pages 14, 37, 38, 40 and 72.)

[26] L. Grigori, J. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 29. IEEE Press, 2008. (Cited on pages 14, 32, 37 and 72.)

[27] M. C. Yeung and T. F. Chan. Probabilistic analysis of Gaussian elimination without pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(2):499–517, 1997. (Cited on page 16.)

[28] D. S. Parker. Random Butterfly Transformations with Applications in Computational Linear Algebra. Technical Report CSD-950023, Computer Science Department, UCLA, 1995. (Cited on pages 16 and 49.)

[29] D. S. Parker and B. Pierce. The randomizing FFT: an aternative to pivoting in Gaussian elimination. Technical Report CSD-950037, Computer Science Department, UCLA, 1995. (Cited on pages 16 and 49.)

[30] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. *ACM Trans. Math. Softw.*, 39(2), 2013. (Cited on pages 16, 17, 32, 45, 49, 50, 52, 65, 72 and 73.)

[31] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002. Second edition. (Cited on pages 16, 26 and 32.)

[32] P. M. Kogge. *The architecture of pipelined computers*. CRC Press, 1981. (Cited on page 18.)

[33] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. (Cited on page 18.)

[34] E. E. Johnson. Completing an MIMD multiprocessor taxonomy. *ACM SIGARCH Computer Architecture News*, 16(3):44–47, 1988. (Cited on page 18.)

[35] R. B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, 1995. (Cited on page 19.)

[36] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The visual instruction set (VIS) in UltraSPARC. In *Computer Conference, IEEE International*, pages 462–462. IEEE Computer Society, 1995. (Cited on page 19.)

[37] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *Micro, IEEE*, 16(4):42–50, 1996. (Cited on page 19.)

[38] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale. Altivec extension to PowerPC accelerates media processing. *Micro, IEEE*, 20(2):85–95, 2000. (Cited on page 19.)

[39] Intel. *Math Kernel Library (MKL)*. `http://www.intel.com/software/products/mkl/`. (Cited on pages 19, 27, 39 and 65.)

[40] Enhanced Intel. Speedstep® technology for the Intel® Pentium® M Processor, 2004. (Cited on page 20.)

[41] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005. (Cited on page 20.)

[42] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002. (Cited on page 20.)

[43] W. Stallings. *Computer Organization and Architecture - Designing for Performance (7. ed.)*. Pearson / Prentice Hall, 2006. (Cited on pages 20 and 22.)

[44] B. I. Witt. M65mp: An experiment in OS/360 multiprocessing. In *Proceedings of the 1968 23rd ACM National Conference*, ACM '68, pages 691–703, New York, NY, USA, 1968. ACM. (Cited on page 20.)

[45] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. (Cited on page 21.)

[46] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 55–55. ACM, 2001. (Cited on page 22.)

[47] A. Moravánszky. Dense matrix algebra on the GPU. *ShaderX2: Shader Programming Tips and Tricks with DirectX*, 9:352–380, 2003. (Cited on page 22.)

[48] CUDA Nvidia. Compute Unified Device Architecture programming guide. 2007. (Cited on page 23.)

[49] T. Ni. Direct Compute–Bring GPU computing to the mainstream. In *GPU Technology Conference*, 2009. (Cited on page 23.)

[50] Khronos Opencl and A. Munshi. The opencl specification version: 1.0 document revision: 48, 2008. (Cited on page 23.)

[51] *TOP500 Supercomputer Site*. `http://www.top500.org`. (Cited on pages 23 and 26.)

[52] J. Dongarra and P. Luszczek. Linpack benchmark. In *Encyclopedia of Parallel Computing*, pages 1033–1036. Springer, 2011. (Cited on pages 23 and 26.)

[53] Intel. *Intel® Xeon Phi™ Coprocessor System Software Developers Guide*. 2012. http://software.intel.com/en-us/articles/. (Cited on page 24.)

[54] G. Chrysos. Intel® Xeon Phi™ Coprocessor-the Architecture. *Intel Whitepaper*, 2014. (Cited on page 24.)

[55] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013. (Cited on page 24.)

[56] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu. Test-driving Intel Xeon Phi. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 137–148. ACM, 2014. (Cited on page 24.)

[57] D. W. Sweeney. An analysis of floating-point addition. *IBM Systems Journal*, 4(1):31–42, 1965. (Cited on page 26.)

[58] International Business Machines Corporation. *System/360 Scientific Subroutine Package (360A-CM-03X) Version II, Programmer's Manual*. IBM Technical Publications Department, White Plains, NY, 1967. (Cited on page 26.)

[59] B. S. Garbow. EISPACK-a package of matrix eigensystem routines. *Computer Physics Communications*, 7(4):179–184, 1974. (Cited on page 26.)

[60] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. (Cited on page 26.)

[61] R. J. Hanson, F. T. Krogh, and C. L. Lawson. A proposal for standard linear algebra subprograms. *ACM Signum Newsletter*, 1973. (Cited on page 26.)

[62] J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*, volume 8. SIAM, 1979. (Cited on pages 26 and 31.)

[63] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003. (Cited on page 26.)

[64] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):18–32, 1988. (Cited on page 26.)

[65] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, March 1988. (Cited on page 26.)

[66] J. Dongarra, J. Cruz, S. Hammerling, and I. S. Duff. Algorithm 679: A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Trans. Math. Softw.*, 16(1):18–28, March 1990. (Cited on page 26.)

[67] J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990. (Cited on page 26.)

[68] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. (Cited on pages 26, 31, 45 and 64.)

[69] AMD. AMD Core Math Library (ACML). *URL http://developer. amd. com/acml. jsp*, 2012. (Cited on page 27.)

[70] IBM Corporation. IBM Parallel Engineering and Scientific Subroutine Library. Guide and Reference. (GC23-3836), 1995. (Cited on page 27.)

[71] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. URL : `http://www.netlib.org/lapack/lawns/lawn131.ps`. (Cited on page 27.)

[72] K. Goto. GotoBLAS. *Texas Advanced Computing Center, University of Texas at Austin, USA. http://www. otc. utexas. edu/ ATdisplay. jsp* , 2007. (Cited on page 27.)

[73] Z. Xianyi, W. Qian, and Z. Chothia. OpenBLAS. *http://xianyi. github. io/ OpenBLAS*, 2012. (Cited on page 27.)

[74] CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15, 2008. (Cited on page 27.)

[75] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: hybrid GPU accelerated linear algebra routines. In *SPIE Defense, Security, and Sensing*, pages 770502–770502. International Society for Optics and Photonics, 2010. (Cited on page 27.)

[76] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. (Cited on pages 27, 31 and 35.)

[77] J. Gunnels, F. Gustavson, G. Henry, and R. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001. (Cited on page 27.)

[78] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 2013. Accepted pending minor modifications. (Cited on page 27.)

[79] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA users' guide. Technical report, Technical report, ICL, UTK, 2009. (Cited on pages 27, 64 and 74.)

[80] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory, 2011. (Cited on pages 28 and 74.)

[81] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi GPUs. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010. (Cited on pages 28 and 50.)

[82] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5&6):232–240, 2010. (Cited on pages 28, 31, 38, 50 and 64.)

[83] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12):645–654, 2010. (Cited on pages 28 and 50.)

[84] J. Choi, J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 107–114. Springer, 1996. (Cited on page 31.)

[85] M. Baboulin, J. Demmel, J. Dongarra, S. Tomov, and V. Volkov. Enhancing the performance of dense linear algebra solvers on GPUs in the MAGMA project. *Poster at Supercomputing*, 8, 2008. (Cited on pages 31 and 64.)

[86] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA'08)*, volume 6126-6127 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008. (Cited on pages 31 and 64.)

[87] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-Avoiding QR decomposition for GPUs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 48–58. IEEE, 2011. (Cited on page 32.)

[88] J. Kurzak and J. J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. *LAPACK Working Note 178*, September 2006. (Cited on pages 33 and 38.)

[89] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 1–10. Springer, 2007. (Cited on page 37.)

[90] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurr. Comput. : Pract. Exper.*, 20:1573–1590, 2007. (Cited on page 37.)

[91] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997. (Cited on page 39.)

[92] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszcsek. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. *Concurrency and Computation: Practice and Experience*, 26(7):1408–1431, 2014. (Cited on page 40.)

[93] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. In *International Conference on Computational Science (ICCS 2012)*, volume 9 of *Procedia Computer Science*, pages 17–26. Elsevier, 2012. (Cited on pages 47, 68, 72 and 73.)

[94] D. Becker, M. Baboulin, and J. Dongarra. Reducing the amount of pivoting in symmetric indefinite systems. In Roman Wyrzykowski et. al., editor, *9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011)*, volume 7203 of *Lecture Notes in Computer Science*, pages 133–142, Heidelberg, 2012. Springer-Verlag. (Cited on page 49.)

[95] M. Baboulin, D. Becker, and J. Dongarra. A parallel tiled solver for dense symmetric indefinite systems on multicore architectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 14–24. IEEE, 2012. (Cited on page 49.)

[96] M. Baboulin, D. Becker, G. Bosilca, A. Danalis, and J. Dongarra. An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems. *Parallel Computing*, 40(7):213–223, 2014. (Cited on page 49.)

[97] J. Dongarra, M. Gates, A. Haidar, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov. Portable HPC programming on Intel many-integrated-core hardware with MAGMA port to Xeon Phi. In *Parallel processing and applied mathematics*, pages 571–581. Springer, 2014. (Cited on page 50.)

[98] A. Haidar, P. Luszczek, S. Tomov, and J. Dongarra. Heterogenous Acceleration for Linear Algebra in Multi-coprocessor Environments. In M. Daydé, O. Marques, and K. Nakajima, editors, *High Performance Computing for*

Computational Science – VECPAR 2014, volume 8969 of *Lecture Notes in Computer Science*, pages 31–42. Springer International Publishing, 2015. (Cited on page 50.)

[99] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *IJHPCA*, 29(2):193–208, 2015. (Cited on page 62.)

[100] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011. (Cited on page 63.)

[101] C. Lameter. Local and remote memory: Memory in a Linux/NUMA system. In *Linux Symposium (OLS2006), Ottawa, Canada*, 2006. `ftp://ftp.tlk-l.net/pub/linux/kernel/people/christoph/pmig/numamemory.pdf`. (Cited on page 64.)

[102] R. Iyer, H. Wang, and L.N. Bhuyan. Design and analysis of static memory management policies for cc-NUMA multiprocessors. *Journal of systems architecture*, 48(1):59–80, 2002. (Cited on page 64.)

[103] A. Kleen. A NUMA API for linux. Technical report, Novel Inc, 2004. `http://www.halobates.de/numaapi3.pdf`. (Cited on pages 64, 65 and 67.)

[104] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010. (Cited on pages 64 and 65.)

[105] J. Gonzàlez-Domìnguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño. Automatic mapping of parallel applications on multicore architectures using the servet benchmark suite. *Computers & Electrical Engineering*, 38(2):258 – 269, 2012. (Cited on page 64.)

[106] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):113 – 121, 1996. (Cited on page 64.)

[107] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, 1989. (Cited on page 64.)

[108] E. Anderson and J. Dongarra. *Evaluating Block Algorithm Variants in LAPACK*. University of Tennessee, Department of Computer Science, April 1990. (Cited on page 64.)

[109] A. Srinivasa and M. Sosonkina. Nonuniform memory affinity strategy in multithreaded sparse matrix computations. In *Proceedings of the 2012 Symposium*

on High Performance Computing, HPC '12, pages 9:1–9:8, San Diego, CA, USA, 2012. (Cited on page 67.)

[110] A. Srinivasa, M. Sosonkina, P. Maris, and J.P. Vary. Efficient Shared-array Accesses in Ab Initio Nuclear Structure Calculations on Multicore Architectures. *Procedia CS*, 9:256–265, 2012. (Cited on page 67.)

[111] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation. *ADVANCED CONFIGURATION AND POWER INTERFACE SPECIFICATION 4.0a*, April 2010. http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf. (Cited on page 67.)

[112] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000. (Cited on page 72.)

[113] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki. On algorithmic variants of parallel Gaussian elimination: Comparison of implementations in terms of performance and numerical properties. Technical report, Innovative Computing Laboratory, University of Tennessee, jul 2013. University of Tennessee Computer Science Technical Report (also LAWN 280). (Cited on page 73.)

[114] A. Rémy, M. Baboulin, M. Sosonkina, and B. Rozoy. Locality optimization on a NUMA architecture for hybrid LU factorization. In *International Conference on Parallel Computing (PARCO 2013)*, volume 25 of *Advance in Parallel Computing*, pages 153–162. IOS Press, 2014. (Cited on page 74.)